UTILIZING BACKWARD ERROR RECOVERY TO ACHIEVE FAULT TOLERANCE IN A SIMD SUPERCOMPUTER

Tamás Bartha [1]

Computer and Automation Research Institute
Hungarian Academy of Sciences

Abstract: APEmille is the third generation of the APE family of supercomputers, but the first one to include built-in support for fault tolerance. The reasons that led to the consideration of reliability issues come in a large part from the experiences with the previous generations. The APE supercomputers are number-crunching engines aimed at solving complex scientific problems. The typical duration of a computation ranges from several days to several weeks. Due to the large number of components built into an APEmille machine, the expected MTTF without fault tolerance may fall into the same range. Thus, a long-running computation could be invalidated by a system failure with high probability, and there would be no guarantee that the results of a successfully terminated program are correct. To improve on this situation the designers of APEmille decided to incorporate fault tolerance into the system. They chose the error removal based approach, composed of error detection, system-level fault diagnosis, system repair, and backward error recovery. This paper presents a failure model of the APEmille components and develops a comprehensive recovery scheme for the whole computer, including even the reliable storage that is used to record the recovery information.

Keywords: backward error recovery, massively parallel systems, SIMD architecture, checkpointing, message logging, reliable distributed storage, replica management

1. INTRODUCTION

This paper describes the proposed backward error recovery mechanism designed for the APEmille parallel computer. APEmille is the latest generation of the APE family of supercomputers. The APEmille project aims at the development of a number crunching machine optimized for *Lattice Gauge Theory* simulations. The machine is an evolution of APE100 (Quadrics) computer, a versatile parallel processor capable of delivering 100 GFlops peak performance in a 2048-node configuration. The new machine has roughly one order of magnitude higher peak computational performance than the preceding generation (scalable from 4 GFlops to 1 TFlops). It also incorporates two main architectural enhancements: *local addressing* capability at each processing element, and independent partitioning of the system to concurrently run different user programs. The APEmille computer is now next to the assembly and test stage before the final release and its controlling software, the APEOS operating system, is also under extensive development.

Even in such a high-performance environment, the amount of computation involved in solving the targeted scientific problems may require several days or weeks of continuous, uninterrupted operation. On the other hand, the complex struc-

[1] The work presented in this paper was carried out in the IEI-CNR, Pisa, with the kind support of Prof. Piero Maestrini

ture and large number of devices constituting the system makes it error-prone, independent on the excellent quality of the components used. Without built-in fault tolerance one may expect multiple fault occurrences during a single job execution. For this reason, APEmille was chosen as an ideal test bed for self-diagnosis and error recovery strategy. Following this idea, the APEmille processors were designed to incorporate both customary fault detection and correction features, and additional hardware support for system-level processor testing as needed by the diagnosis algorithm. Although the APEmille self-diagnosis theory and hardware support is already mature, well formulated, and tested both by simulation and in practice (Maestrini and Santi, 1995; Chessa *et al.*, 1999), error recovery has not yet been elaborately studied. This paper is the first attempt to form a comprehensive recovery concept for APEmille.

In the rest of this paper we briefly introduce the basic hardware architecture of the APEmille machine, then describe the fault diagnostic procedure and the built-in hardware support for diagnosis. The failure model and the assumptions on the failure semantics of different system components are presented. Based on this knowledge we explain the decisions behind the chosen error recovery approach. The recovery strategy for each component is outlined, giving alternatives for tolerating single or multiple component faults. We mention the techniques used in the recovery subsystem, and consider the problem of providing a reliable storage to record the recovery information. For each proposed solution the implementation-related consequences of APEmille's unique hardware and software architecture are examined. In the closing part of the paper we give directions for further research in the area.

1.1 *Hardware overview*

From the user's point of view the APEmille is a parallel computer consisting of processing elements (PEs) optimized for floating-point arithmetic. The PEs are arranged in a three-dimensional mesh topology with toroidal wrapped-around interconnections in all the three directions. APEmille has a Single Instruction Multiple Data (SIMD) architecture. Every PE executes exactly the same instruction in each step of the computation, but they have private memory banks, thus each node can operate on its own specific data. Moreover, the SIMD paradigm is supplemented with many enhancements, like the *broadcast* and *soft* routing methods providing data exchange between two arbitrary nodes, and a *local addressing* feature: all PEs may access their local memory with their own local address (Aglietti and et al. (The APEmille Collaboration), 1997). A large APEmille machine can be split up into several, logically independent partitions, this way multiple users can use it simultaneously.

On the hardware level, the system structure is more complicated. It is built up of three main functional components. In addition to the application processors (referred to as *Jmille*), there is a central processing unit (called *Tmille*) performing mainly flow-control, signal handling, and global integer operations; and a communication controller (called *Cmille*) functioning as an interface between Jmille, Tmille, and the rest of the system. The architecture is divided into three, hierarchically structured levels. On the lowest level there is the smallest independent functional unit, called the *Processing Board* (PB) or *cluster*. A PB contains eight Jmille processors, one Tmille control processor, and one Cmille commuter. The arrangement of a Processing Board can be seen in Figure 1.

Jmille is the processing element in the APEmille system. It supports arithmetic operations on both integer and floating point data words. The operands and/or the result are located in a large register file (RF) composed of 512 registers. Each Jmille node has a local memory for its own exclusive use. Memory addresses can be global, in which case they are generated by the Tmille processor and sent to every Jmille on the same PB; but global addresses can also be modified with a value local to the nodes. Tmille also has its own data memory, an integer ALU, and an Address Generation Unit (AGU). All Tmille processors belonging to the same partition execute the same instruction stream and are initialized with a common set of values, i.e., the whole partition behaves like an independent SIMD computing engine. The Cmille commuter has both a simple topological structure and a restricted set of capabilities. Each Cmille is connected to the eight Jmille units and the Tmille residing on its PB. Additionally, it is connected to the six nearest neighbor commuters in the three spatial directions. The Jmille, Tmille, and Cmille devices are built into custom designed integrated circuits.

The next hierarchy level above Processing Boards is called an *APE Unit*. APE Units are built of four PBs, and are connected to an external, standalone computer called *Local Host* (LH). The host computers act as supervisors over the attached PBs and provide local disk storage services. The hosts can read and modify the memory and registers of the APEmille processors. On the other hand, Tmille can trigger interrupts to signify exceptions, service requests, or local conditions. These signals are handled by the *Root Board* (RB)

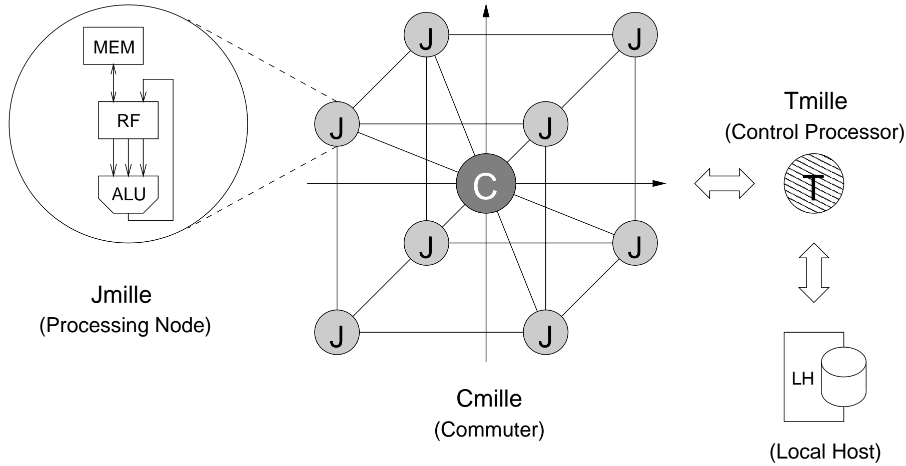

Fig. 1. The logical structure of a Processing Board

built in the LH. The host computers have a PC-compatible architecture, and incorporate a local disk unit on which they store the local portion of the operating system and application-specific data. The interconnection between PBs is routed via a synchronous network managed by the Cmille commuters on the boards.

A configuration of four APE Units is called a *crate*, and it is housed in a standard rack (see Figure 2). The rack contains a custom Compact PCI backplane with four APE Units. There are 16 APEmille PBs on one side of the backplane, and 4 Local Host Boards with supplemental Service Boards (Control Network controller cards, SCSI controllers, Fast Ethernet cards) on the other side. The removable physical assemblies, i.e., the *replaceable units* (RUs) are the Processing Boards, the Local Host Boards, and the Service Boards. The hosts of different APE Units are cooperating over a general-purpose, high-performance Control Network. Also attached to this network is a special host computer acting as the *Global Host* (GH) of the entire APEmille machine. The GH manages the global disk storage and collects the global signals like halt requests, exceptions, `if` conditions, etc. Several crates can be stacked on top of each other, until the system reaches its maximum configuration of 4096 ($64 \times 8 \times 8$) processing elements.

The APEmille machine has two main operating modes: `run` mode and `system` mode. On power-up the machine enters the `system` mode. The processing units are frozen, and the LHs can access the complete register file and address space of the Jmille, Tmille and Cmille devices. The OS first loads the program code and data in the memory and sets the proper registers to initial values, then generates a transition from system to `run` mode. Once in `run` mode, the processing hardware starts executing the program. The two operating modes are supplemented by a special diagnostic mode, called the `equal` mode. The transition between the operating and diagnostic modes can be initiated by the hardware, by the operating system, and from the user console; but the application program can also trigger this transition by a dedicated low-level instruction. These software-generated transitions to `system` mode are called *traps*, they are useful for signaling service requests towards the Host computers.

1.2 *Hardware support for diagnosis*

A novel feature of the APEmille computer is that low-level diagnostics support is integrated in the system. The testing of the Jmille, Tmille, and Cmille units is done user-transparently on the hardware level. The main testing mechanism employed is *comparison*. For this purpose self-checking comparators are integrated in the circuitry of the Cmille communication processor. The comparators have a self-checking design that detects single stuck-at comparator faults. The three basic components of the Processing Boards are tested separately.

Pairs of Tmille processors or pairs of Cmille commuters can be tested by comparing the output sequences they write on the bus. These units are controlled in a uniform way and perform the same sequence of instructions. They operate on the global variables and the instruction flow process identical data, which are identical due to the SIMD execution model. For this reason, the testing of the Tmille and Cmille units can proceed concurrently with the computation assigned to the machine.

Although the Jmille units also execute identical instructions, they normally work with local data that varies for each node. Therefore, the testing of Jmille processors is carried out in `equal` mode, in a special *diagnostic session*. The diagnostic session is composed of two phases, called the *Local Equal* and the *Remote Equal* phase. In the Local Equal phase the Jmille units are loaded with the

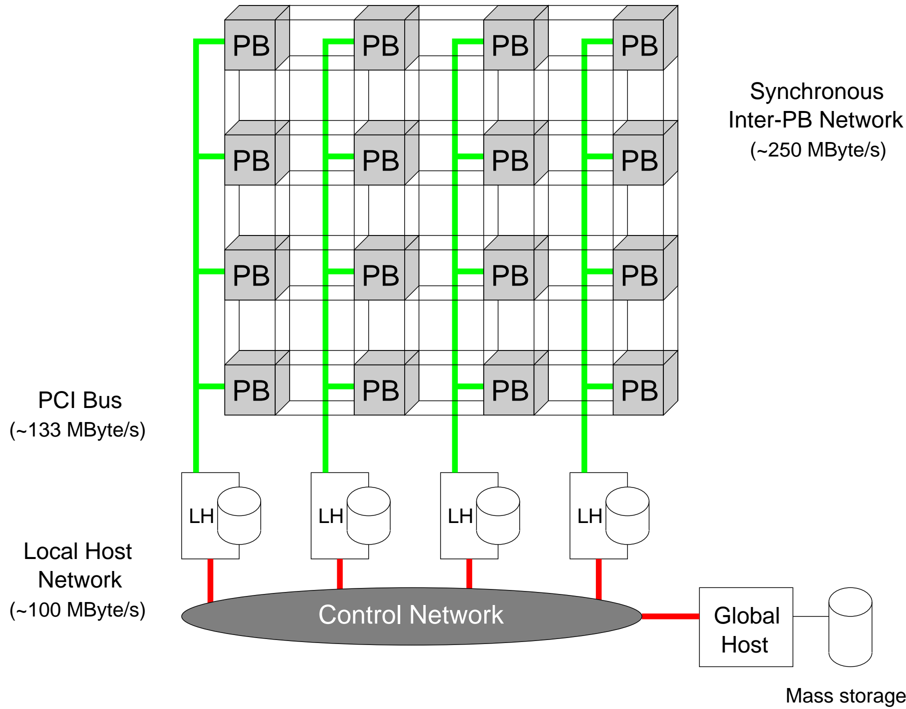

Fig. 2. The APEmille crate

same set of instruction and memory values. During test program execution each Jmille memory access and instruction fetch is routed through the Cmille comparators. In this way, the eight Jmille processors of the same PB are pair-wise compared along a logical ring. The comparison results are processed by a system-level diagnosis algorithm, producing a preliminary classification of the processor fault states. This preliminary diagnostic image is extended and refined in the Remote Equal phase. For each pair of adjacent Processing Boards two Jmille processors are elected to be *witnesses*. The two witness Jmille nodes are compared by a pair of comparators located in the Cmille commuters of the adjacent boards. The so obtained global comparison syndrome is further analyzed by the diagnosis algorithm (Chessa *et al.*, 1999), and finally every Processing Board is labeled as fault-free, faulty (or partly faulty), or suspect (when the diagnosis is incomplete and the state of some PBs cannot be decided).

1.3 *Failure model and assumptions*

Any reasoning about recovery techniques is possible only if one has a well-defined model of the failure behavior of APEmille components. Given this model, the most suitable methods of backward error recovery can be selected, their applicability and characteristics can be verified. This section presents an informal failure model of APEmille components, which grabs specifically those aspects that need to be taken account in planning the recovery actions. The model only serves as a basis for selecting the fault tolerance mechanisms. It was derived from the system specification by purely theoretical methods, at a later stage it shall be refined when practical experience and measurement data will be available.

The attentive reader has surely noted, that APEmille has a peculiar two-fold nature (Aglietti *et al.*, 1997). On the one hand, it contains a computational engine with its uncommon structure of Tmille, Cmille and Jmille processors. Further on we will refer to this part of the machine as the *SIMD part*. On the other hand, it employs a network of host workstations for delivering services and controlling and monitoring the SIMD part. These computers can be viewed as a special case of the Multiple Instruction Multiple Data (MIMD) class of parallel systems, hence we will refer to them as the *MIMD part* of APEmille. The two-fold nature also reflects in the operating modes: in `run` mode the SIMD part is active and executes the user applications, while in `system` mode it is frozen and the MIMD part delivers the requested services or supervises the machine.

We inherit the modeling methodology of Lampson (Lampson *et al.*, 1981). The main classes of components considered in our model are processors, memory, disk storage, and the communication network. Based upon this general model, first we account for the failure characteristics of the components in the SIMD part.

Jmille and Tmille processors conform to the general processor model with the restriction that they execute only a single task. Therefore, these proces-

`run` mode				
Component	**Test method**	**Failure semantics**	**Undetected**	**Recovery**
Jmille processor	comparator	value	common-mode, transient	checkpoint
Tmille processor	comparator	value	common-mode	checkpoint
Cmille commuter	comparator	omission/value	common-mode	checkpoint
memory	EDAC	omission/value	n-bit ($n > 2$)	overwrite
(inter-PB) network	EDAC	performance	n-bit ($n > 2$)	retransmission
`system` mode				
Component	**Test method**	**Failure semantics**	**Undetected**	**Recovery**
(Host) processor	watchdog timer	crash (fail-stop)	value	message logging
(Host) memory	parity	omission/value	n-bit ($n > 1$)	overwrite
(Host) disk	EDAC	omission	multiple bit	duplication
(Control) network	checksum	performance	value	retransmission

Table 1. Failure model of system components

sors can be modeled as a deterministic *finite state automaton* (FSA). We do not consider arbitrary or Byzantine failure semantics for these processors; such failure behavior is quite improbable and hard to handle due to the special architecture. Halting, omission and performance failure behaviour implies that the processor does not drive the bus in time, leaving the bus signals in an arbitrary state. Due to the asynchronous interface the external observer sees it as a *value* failure. We assume that all of the permanent value failures are detected by the mutual comparison testing of the processors. *Common-mode* failures remain undetected after testing, but they are uncovered at a later stage by the diagnostic algorithm. Transient value failures of Jmille units, however, cannot be detected during `run` mode.

On investigating processor interactions, we assume that a Jmille processor cannot make another Jmille to fail (in the physical sense), since there is no direct connection between them. The communication is memory mapped: a special address signifies a remote access to the memory of another Jmille processor. A value failure during a memory write operation can corrupt the local or the remote Jmille memory contents, causing cumulative errors. There are three possible erroneous memory write operations respective to the failure affecting the address and/or the value of the written data. Of these three, we model the 'invalid data to right address' action as a *bad write* error, the 'valid/invalid data to wrong address' actions as the combination of a bad write (to the good address) and a spontaneous *decay* (at the wrong address) errors.

Tmille processors are also indirectly connected, so they cannot make each other to fail similarly to Jmilles. There is also a very low probability of them causing cumulative errors in Jmille processors by sending wrong instruction codes/invalid global addresses, because apart from common-mode faults these are detected by comparison testing. The memory of a Tmille processor contains only programs. By ruling out self-modifying programs we can assure that this memory will never be written. Thus, 'bad write' errors become impossible, leaving 'decays' to be the sole source of Tmille memory errors.

The Cmille commuter is the intercessor among the Jmille and Tmille processors, and the adjacent Processing Boards. Its value failures may produce cumulative errors similarly to Tmille processors by delivering incorrect data/instructions to Jmilles. However, transmission value failures caused by transient or permanent faults in the interconnection network are either covered by the employed *error detecting and correcting* (EDAC) code, or manifest themselves as 'bad write' or 'decay' errors of the destination memory component. Furthermore, crash failures during remote communication are supposed to be detected by the other involved Cmille commuter, which in this case raises an exception and—being unable to deliver the data—creates an omission failure. We do not account for the fault in the comparators used for testing (although they are also a part of the Cmille circuitry), this duty is left to the fault diagnosis algorithm.

The processors of the Global and Local Hosts fully comply with the general processor model of Lampson. Additional considerations can be drawn from the fact that these computers employ a multitasking, multi-user operating system. The OS (with hardware support) isolates different processes from each other, and prevents accessing the resources assigned to other processes in an unauthorized way. It provides primitives for synchronization and mutual exclusion to support the interaction of processes and sharing of resources. On this basis, we assume that a failed process will eventually raise an exception (either due to an undesired condition such as division-by-zero or a protection violation) and gets stopped, i.e., the hosts processes have a fail-stop failure semantics. During the crash, processes cannot interfere with the local state of another process, but can (and probably will) make their own resources inconsistent or corrupted. Processors are composed of a

set of processes and hence behave similarly: crash on error, and after a period of time become fail-silent. The halted state can be detected using a hardware or software watchdog timer, and in the case of communication by time-out on periodically sent diagnostic <I'm alive> or *heartbeat* messages.

The memory components in both the SIMD and MIMD part employ an EDAC code to detect (and possibly correct) read errors. In the case of Jmille and Tmille memories a modified Hamming code is used, capable of detecting all double-bit (and some triple-bit) errors and correcting all single-bit errors. Corrected errors are transparent to the user and so they cannot be considered as failures. Yet, the number of corrected errors is counted in a special register, because frequent recurrence of single-bit errors might indicate an underlying permanent or intermittent fault as the cause. The host computers add a single parity bit to each word of memory, detecting all single-bit errors (and some multiple bit errors as well). Detected but uncorrectable errors raise an exception and are noted by the operating system, however, the memory contents are lost and cannot be recovered. Thus, the memory exhibits *omission* failure semantics. Undetected read errors become *value* failures.

The communication in the synchronous inter-PB network is also guarded by EDAC. The asynchronous Control Network uses a commodity network protocol. The validity of messages is controlled by various checksum schemes in several protocol layers. Lost or incorrectly delivered messages of the SIMD part are transformed into 'bad write' or 'memory decay' errors due to the memory mapped nature of the communication. In the Control Network lost messages are assumed to be detected by the delivery control mechanisms and retransmitted until it arrives successfully. Undetected value failures in the messages are disasters. Consequently, both communication networks are modeled as having a *performance* failure semantics.

Disk storage can be found only in the Global and Local Host computers. Its model is quite similar to the model of the memory components. Additionally, on the basis of the advanced manufacturing technology and the combination of EDAC and checksums transparently managed by modern hard disks, we assume that all value failures will be detected. Moreover, a large part of them will be corrected transparently by the disk controller hardware, notifying the OS to avoid the unreliable disk area by adapting its file allocation strategy. Should this assumption fail, the user can substitute a redundant disk array (RAID) in place of a single hard disk to make the assumption true. Detected but uncorrectable read errors manifest themselves as *omission* failures.

2. ERROR RECOVERY

There are two main options in recovering from errors. *Backward error recovery* chooses a known valid state from the system's past. This is accomplished by periodically saving the complete or partial system state to a reliable medium (called the *stable storage*) during the failure-free execution. Then, after an error occurrence the tasks are *rolled back* to the previous valid state using the information stored on the stable storage. *Forward error recovery* obtains a new valid state that is not included in the history of the system. This approach is more efficient than backward error recovery, because a history of the valid system states is not needed, and the recovery process performs valuable work while creating the new consistent system state. Unfortunately, forward error recovery is also quite application-specific, and since APEmille is a general purpose computer, rollback recovery remains as the only useful alternative.

Backward error recovery attempts to maintain the *consistency* of a computation by returning the system to a previous state upon an error occurrence. Two basic services are used in this process (Chandy and Lamport, 1985): *state recording* which stores global system states to a reliable medium, and *state restoration* which assembles the stored information and recreates a consistent global restored state. These services can be either *semi-automatic* if they are executed upon the invocation of the application programmer, or *user-transparent* if the necessary actions are carried out automatically, without any user interaction. Recovery techniques can be classified in three major classes according to the characteristics of the above two services:

Checkpointing. Checkpointing saves a copy of the application state (and optionally the state of the communication channels). Upon a failure, the checkpoint can be used to restore a previous, failure-free state of a failed process. *Independent* checkpointing methods take the checkpoints process-wise. In this case, there is no guarantee that a consistent global state can be restored from the stored set of process states. The state restoration service must be able to determine which the most recent recovery line (also called as *maximum recoverable global state*) included among the set of stored local checkpoints. This makes the recovery process more complicated, and there is the possibility of *domino effect*. *Consistent* checkpointing methods avoid these disadvantages by

coordinating the checkpointing actions among the processes to guarantee that the stored set of local checkpoints is always a recovery line.

Message logging. In message logging the interactions between processes (denoted as events) are saved instead of the complete global state of the system. Events drive the transition of a process from one state to another, thus determine the course of computation. In a *piecewise deterministic* system events are limited to the sending and receiving of messages. Recording of the relevant information about messages is called *logging*. The recovery protocol must recreate the exact order and content of each message transmitted before the failure occurred. Storing the contents of messages in the log speeds up the recovery process. Yet, it is not strictly necessary for a successful rollback, since the messages can be regenerated together with the other constituents of the distributed computation.

Hybrid techniques. This is a composition of the previous two techniques which combines their advantages. Although it is possible to implement backward error recovery solely based on message logging, the stored log data could grow unacceptably large over a long time span. Therefore it is customary to combine logging with checkpointing: when a global state becomes stable among the recorded checkpoints, log entries related to messages preceding the global state are no longer meaningful, thus can be discarded.

The fault tolerance techniques proposed for the implementation of backward error recovery in the APEmille computer are summarized in Table 2. For each main component class of APEmille we consider the cases when a single, multiple, or all components in the class fail, and suggest the most appropriate method accordingly. The chosen methods are explained and briefly described in the following sections (for more information on recovery techniques see (Bartha, 1998)).

2.1 *APEmille processors*

For the processors in the SIMD part the most suitable recovery mechanism is *checkpointing*. This choice is supported by the following arguments:

- It is difficult to implement message logging for the processing elements. The communication among the PEs is memory mapped. The PE initiating the transfer knows when a remote access takes place, but the other participant of the communication is the remote memory, and the owner processor is not notified explicitly of the read or write operation that affected his memory. For this reason, it would be complicated to realize the acknowledgement and bookkeeping schemes most message logging protocols require from the recipient processor.
- The state of the PEs changes quickly, and in a large extent. The applications are assumed to frequently access all elements of large data sets. There are many operations that affect the local and remote states; most of them are not (and cannot be) implemented as atomic transactions. Therefore, message logs would be updated often and would grow quite huge. The applications would be suspended frequently during the writing the message logs to stable storage (even if optimistic or causal message logging is used). The local and remote states are expected to have a complex interdependence, making it even more cumbersome to maintain the message logs.
- Since there is only a single process running on the PEs, logging the messages user-transparently is only possible by compiler-assisted inserting of the bookkeeping code in the user programs. This would make the executable parts of the applications grow larger.

The implementation of checkpointing is straightforward. The checkpointing or recovery procedures are invoked after the diagnostic session, when machine operation has been switched into `system` mode. Due to the transition to `system` mode, there is no ongoing computation or communication in the SIMD part, and the host computers have complete control over the memory and register-file areas of the APEmille processors. There is no need for checkpoint coordination and it is sufficient to store the state of the application without the state of the communication channels, since all data transfers have been carried out before the checkpointing procedure was invoked.

The checkpoint must contain the value of all variables and data structures that describe the actual state of the application process. Hence, it is up to the system or application programmer to decide what must be included in the set of checkpointed information. The most simple implementation of checkpointing saves the whole address space that belongs to the given process. Note, that this implies a huge amount of data in the case of APEmille: all of the local memories belonging the the 8×4 Jmille nodes of an APE Unit must be stored on the disk of the supervising Local Host computer. Starting from an full initial checkpoint there are methods that reduce the extent of later checkpoints, provided the state of the process changes "moderately." The techniques we propose are based on coding techniques. In the following we describe mention methods which store a reduced checkpoint generated by error detection and correction (EDAC) encoding of the data, and

Component	Number of faulty components		
	One ($k = 1$)	Some ($1 < k \ll n$)	(Nearly) all ($k \approx n$)
Jmille processor	parity checkpointing	EDAC checkpointing	complete state saving
Tmille processor	checkpointing		
Cmille commuter	checkpointing		
(host) processor	sender-based logging	family-based logging	
(host) disk	available copy replication		–

Table 2. Proposed fault tolerance techniques

exploit the additional failure information provided by system-level diagnosis of the application processors.

Parity checkpointing uses $n + 1$ parity instead of a full replica to maintain the checkpointed global state (Plank, 1993). The mechanism of parity checkpointing is as follows: the respective memory locations (the memory bits in the same address and position) of the n parallel processors are treated as an n-bit group. Every group of n memory bits is supplemented by a single parity bit, and the resulting parity bit array is stored on a central reliable storage medium. If a single-bit memory fault occurs, a new array parity is computed using the actual state of the local processor memory contents, and the new array parity is compared with the stored reference array parity. The difference pinpoints the location of the erroneous bit, which can be corrected by a simple inversion.

A significant drawback of parity checkpointing is that it handles only a single node failure. The problem originates in protecting a group of n memory bits with only a single parity bit. In the *EDAC checkpointing* scheme (Bartha, 2000) the parity checkpointing scheme is combined with the multiple-bit fault-tolerant *$P + Q$ Redundancy* employed in RAID Level 6 (Gibson, 1991). The underlying idea is to assign an l-bit long, p-bit error detecting, q-bit error correcting error detecting and correcting code to every group of n memory bits. The generated EDAC code can be used to repair the corrupted memory contents of the failed processors, provided that the number of processors to fail always remains below p. Based on the bit-wise EDAC code the recovery algorithm can determine the *quantity* of the incorrect bits in a certain memory area, and exploit the system-level diagnostic image to find the *location* of errors. The memory contents can be recovered by simply inverting the corrupted bits. To consider the space saving represented by this approach, suppose that the system has n processors with local memories of w words, each having the size of b-bits. Using EDAC checkpointing, a code of $w \times b \times l$ bits is generated and stored in the stable storage, instead of the total state saving which requires $n \times w \times b$ bits. Clearly, the EDAC code saves a significant amount of storage space until $p \ll n/2$. However, large amount of incorrect memory bits (caused for example by an undetected transient value failure propagated to several Jmille units) can only be tolerated by storing the complete global state.

2.2 *Global and Local Host computers*

Unlike the processors in the SIMD part, the suggested recovery mechanism for the host computers is *message logging*. The following arguments justify the preference of message logging over checkpointing in this case:

- There are two kinds of actions that a host computer must perform: serving requests and handling exceptions coming from the Processing Board, and executing *remote procedure calls* (RPCs) of other hosts. These are a few, well-defined, complex operations. Provided they are implemented to be atomic and/or restartable, and their execution order/dependencies are recorded reliably, then it is possible to reset the computer (if needed) and repeat or replay the failed operation(s).
- Since the host computers are devoted to mainly supervising and monitoring tasks, the frequency of service and RPC requests is expected to be relatively low compared to the execution speed of the PEs. Therefore, the size of the message logs will not grow over a reasonable limit during a computation.
- The host computers offer a layered multitasking environment. The operating system consists of many small processes. In the case of checkpointing the state of each process should be recorded separately, while paying special attention not to violate the consistency of the global state these individual process states are supposed to form. Message logging may concentrate only on the interaction of processors via the communication interface leaving the interaction of local processes out of consideration.
- The recovery procedure can be realized as an operating system layer, consisting of a separate process (or a collection of processes). By placing the Recovery Layer low enough in the layer hierarchy, it can capture every incoming service requests and exceptions signals. Then, it can perform the necessary logging and bookkeeping actions, suspend or deliver

the received messages; in other words, it has a complete control over the communication interface. In this way, message logging can be implemented user-transparently. Host computers may even utilize the idle time while the machine is in `run` mode to write the volatile message logs to stable storage.

- The number of host computers is small relative to the number of PEs. Each host is independent from the other, has its own power source and disk storage. Certain run-time transient faults causing detected errors (parity error in memory, access protection violation, etc.) may be handled at the OS level, preventing them to become failures. For these reasons, the MIMD part is assumed to have a higher MUT (Mean Up Time) than the SIMD part. Fault tolerance may take advantage of this, and we can employ techniques that tolerate only a limited number of failed host computers, but run more efficiently than worst-case solutions.

Message logging requires the operations working with system resources to be *atomic*, in order to keep them always in a consistent state. An action is atomic if it has both of two basic properties: it is *unitary* and *serializable*. Analogously, the notion of *atomic transactions* can be introduced. The system guarantees that after a recovery from a crash either all of the commands constituting an atomic transaction will have been successfully carried out, or none of them will have been. Additionally, atomic transactions are indivisible with respect to other transactions that may be executing concurrently; that is their execution is always equivalent to some serial order. Furthermore, there is a third condition (independent on atomicity) which denotes a transaction to be *restartable* if the transaction can be successfully repeated during crash recovery even if it has been in progress when the crash occurred.

Like checkpoints, the message log must also be retrieved reliably, thus it is also stored on stable storage. Log entries are smaller but more often updated than checkpoints, therefore the frequent access to the slow stable storage can create a performance bottleneck. There is a speed/reliability tradeoff between the two main classes of logging algorithms: the *pessimistic* and *optimistic* approach. Pessimistic methods always synchronize message delivery and logging by writing each log entry immediately to stable storage. They guarantee that the logged information is always consistent. On the other hand, optimistic methods favor performance to consistency. They assume that failures are rare, hence they try minimize the logging overhead during normal computation. They record the log entries temporarily in volatile memory, and handle the inconsistencies that may arise due to a failure during recovery. Another design decision is to choose the units that perform the logging action. With *receiver-based* logging, the processes participating in a distributed computation log the messages upon receipt. When recovering from a failure, the failed process restarts from scratch or a previous checkpoint and replays the messages in the log. In *sender-based* logging, messages are stored at the sender process. If a process fails, the messages needed for execution replay are resent by their originator.

Sender-based logging protocols have the favorable property of tolerating a single host failure even in the optimistic setting, when the messages are stored in the volatile memory of the sender process (Johnson and Zwaenepoel, 1987). Due to its unbeatable performance and simple implementation it is the method of choice when the failure rate is low, i.e., only a single Host computer is expected to fail at a time. If multiple simultaneous host failures may occur, an extended version of sender-based logging must be used. This approach known as the *family-based logging* (FBL) protocol (Alvisi and Marzullo, 1995).

The underlying idea of FBL logging is that a message is partially logged by the time is is sent and it must be fully logged by the time it is *relevant*. FBL is an optimal message logging protocol in the sense that it does not send any additional messages over those needed to mask transient link failures. FBL protocols for $f > 1$ failures distribute the implementation of the recovery procedure associated with a process to a larger degree than in the single failure case. For this purpose they need to maintain and disseminate some *dependency data* about partially logged messages. The dependency data contains a mixture of sender- and receiver-based information. This means that a process must not only log the content of all the messages it sends, but it may also be required to log the messages it receives and are received by its ancestors (successors in the communication chain) up to a degree of f.

In the above discussion of rollback recovery techniques the problem of saving the checkpoints and message logs to the stable storage was mentioned many times. In the next section we explain the concept of a stable storage, and outline a practical approach to realizing such a device based on the disk storage units of the APEmille Global and Local Host computers.

3. RELIABLE STORAGE

A reliable device used to store persistent information or intermediate data structures of compound operations is called a *stable storage* (Lampson *et*

al., 1981; Banatre *et al.*, 1988). The two main properties of this abstract device can be summarized as follows:

- Resistance against external hardware or software failures (processor failures, invalid storage accesses) and internal errors such as decays, and
- Atomicity of read and write operations.

The stable storage system we propose for recording checkpoints and other persistent data resides entirely on the disk storage of the Local Host computers. Only a part of the disk area is devoted to this purpose, the rest holds a normal UNIX file system. There is a special process, called the *Storage Manager* (SM), which has an exclusive access to the stable storage area. We assume that the resource protection and process isolation services provided by the OS are adequate to prevent any failed process to corrupt the data in the stable area.

Besides, the storage system must handle the following types of faults: *external hardware faults* such as the erroneous behavior of the host processors, *external software failures* that manifest themselves at the user interface of the Storage Manager, and *internal errors* of the SM process and the disk drive/controller. The stable storage is designed to hold objects that can have the size of a single or multiple memory pages. We are not interested in the internal structure of an object, but we require that a large object of many pages should occupy a contiguous space in the memory. There are two operations defined on the objects: a `write` command to create or update an object in the stable storage, and a `read` command to retrieve the last written value of the object.

The techniques proposed to realize the stable properties are listed in Table 3. Soft internal errors originated in the disk drive and controller are partly handled by the hardware itself; modern storage systems are equipped with powerful coding mechanisms to detect and correct decay errors. To tolerate soft errors during disk access we employ the *careful disk operations*, as described in (Lampson, 1981). A *careful read* operation repeatedly performs a normal disk read until it gets a good status or a predefined limit of retries is exceeded. This eliminates soft read errors. A *careful write* operation repeatedly performs a normal disk write followed by a read until it returns a good status with the data being written. This eliminates null writes and bad writes to good addresses. Address value problems are discovered by the Storage Manager process using access control and a simple semantic check on the objects.

The techniques that help the Storage Manager process to detect the interaction errors at its user interface we inherit from (Banatre *et al.*, 1988). The purpose of these detection techniques are twofold:

(1) they ensure a proper initialization of the transactions by checking that the first access to the object is valid, and
(2) they enforce that the read and write transactions respect their predefined semantics.

The checking of the first access is performed by a key control mechanism. Any transaction that changes an object must first provide a *key*, which is inspected by the SM. For the purpose of the key we suggest to use a checksum, generated from the entire data content of the object. The checksum mechanism should be inexpensive with respect to the computation, but must be capable of verifying the integrity of the object. The checksum is stored together with the object. At the beginning of the transaction the SM compares the stored and the provided checksums. If they match, then the access to the object is granted to the requesting process. When the transaction successfully terminates, the Storage Manager computes a new checksum based the changed data contents and stores it as the new key to the object. On the next access the requesting process must present the new key to gain control of the object.

After passing access control the requested transaction can take place. Interaction problems during the execution of the transaction are detected by checking the semantics of the performed operation. The following two properties characterize the semantics of atomic read and write transactions:

P access: any access to an object implies accessing all of the pages it is composed of once and only once. This property encompasses the unitary ("all or nothing") characteristic of the atomic transactions on the object.

C access: all the pages that constitute an object are accessed in the ascending order of their logical address. This property helps to filter out incorrect accesses to the object.

These properties fit well the purpose of storing checkpoints in our stable storage, since the scientific computations running on APEmille typically use large contiguous data structures such as vectors and arrays. Although the semantic checking power they provide is not very strong, they detect the most common malfunction during the stable storage access (Banatre *et al.*, 1988), the address value failure. And they can be verified quite easily by a simple counter. More powerful semantic checking mechanisms (such as application-specific data acceptance tests) can be developed with a deeper knowledge of a certain application.

In order to implement stability, there must be at least one intact and up-to-date instance of the

Desired property	Requirement	Realization technique
Fault tolerance of hardware/software failures	resilience	careful disk operations
	stability	available copy replication
		Replica Majority Commit
	consistency	access control
		semantic integrity checking
Atomicity of read/update operations	indivisibility	shadow updating
	serializability	two-phase locking (in ACR)
		three-phase commit (in RMC)

Table 3. Realization of the stable storage properties

data deposited in the stable storage at any time. Permanent internal faults and external hardware/software failures can make the disk storage of any host computer unavailable. Therefore, the stable data area must be maintained at multiple host computers. The different copies of the stable data area are called *replicas*, they a supervised by a *replica manager* (in the APEmille the Storage Manager process can fulfill this role) which uses a *replica control protocol* to keep the available copies up-to-date and organize the accesses to the replicated objects user-transparently, as if it would be a single, highly available storage system. Two protocols were selected for APEmille depending on the system configuration: the *available copy replication* (APC) for smaller systems, and the *Replica Majority Commit* (RMC) protocol for complex configurations. Details of these replica control protocols are presented in the next section.

The atomicity of the read/write transactions can be guaranteed by fulfilling two basic conditions. The simplest method for ensuring the unitary or indivisibility property of write transactions is *shadow updating* (Cristian, 1991). This technique allocates two buffers of identical size to record the object. One of the buffers has the actual data of the object, the other one is called the *shadow* buffer. A binary pointer indicates which buffer plays the active/shadow part at a certain point of time. A write operation is carried out in two steps: (1) the new value of the object is written in the shadow buffer; (2) the pointer is inverted, i.e., the role of the active/shadow part is exchanged. If an error occurs during the first step, then the second step is omitted, thus the changes are not reflected in the state of the object. The second step is so simple, that it can be implemented as an atomic action using a read-modify-write (RMW) or test-and-set (TAS) instruction found in most modern processors.

The serializability property means that atomic transactions must always be carried out according to a serial schedule, or in other words, there must exist a partial ordering of read and write events. This property also plays an important role in the multiple copy update problem, therefore serializability is inherent to the replica control protocols mentioned above: available copy replication employs the two-phase locking technique, while Replica Majority Commit contains a modified three-phase commit protocol.

3.1 *Management of the replicated data*

An ideal replication control protocol should guarantee the consistency of the replicated data in the presence of any arbitrary combination of non-Byzantine failures, while providing the highest possible data availability and requiring the lowest possible overhead. Data consistency in a replicated storage system means that the distributed processes accessing the stored information experience an identical view of the global state of the replicated data at any point of time. This notion incorporates two aspects: the *mutual consistency* of different copies, and the *internal consistency* of each copy. Copies of the data are mutually consistent if they are identical; since this is impossible to achieve for every instant of time this constraint is relaxed to require that multiple copies must converge to the same final state as all access activities cease.

Internal consistency of the data refers to the information content and involves both the *semantic integrity* of the stored object and the *atomic property* of the update operations. Semantic integrity stresses the need for the stored data to reflect accurately the state of the real world object it describes. Atomic transactions guarantee that the storage system reflects either none or all of the actions caused by a create/update transaction. Since the transaction is committed only if it does not violate semantic integrity, atomic transactions guarantee the internal consistency of the data, and so does any *serial schedule* of atomic transactions. Thus for a replicated storage with the possibility of concurrent transaction processing, mechanisms must be provided to generate serializable transaction schedules.

In the absence of network partitions the consistency of the replicated data can be ensured by the *available copy replication* (ACR) protocols (Pâris and Long, 1990). These replication control protocols were designed to provide better fault tolerance characteristics than voting methods in

environments that preclude partial communication failures. The operation of ACR protocols is based on: (1 imposing a total ordering on all writes so that all replicas receive these requests in the same order, (2) broadcasting these writes to all available replicas, and (3) requiring that replicas residing on nodes recovering from a failure remain unavailable until they are brought up-to-date. Using this mechanism read requests never need to access more than one available replica, because *all* available replicas are ensured to be valid and contain the latest version of the data. Furthermore, the replicated data can be accessed as long as there is at least *one* available replica.

The situation is quite different when network partitions (i.e., normally connected and logically coherent parts of a network are separated by a communication failure) must be taken into account (Pâris, 1994). This situation may occur in APEmille when two complete configurations are connected together to form an even more potent computing environment. A major limitation of the available copy replication strategy in this case is that it requires *reliable failure detectors*. As network failure detection is usually implemented by time-outs (which is an unreliable failure detection method), partitioning may lead to falsely suspect a correct but inaccessible process. For such a case we propose the use of the RMC (Replica Majority Commit) replication control protocol (Guerraoui *et al.*, 1996). The RMC protocol was introduced to solve the *update majority* problem in the presence of unreliable failure detectors. The update majority problem is the task of updating a replicated object according to the majority voting strategy. Majority voting refers to a replication protocol where a read or write operation on a logical object must always access some majority of the replicas:

- On reading the object: the transaction accesses some majority of the replicas, chooses the one with the highest version number, and returns the contents of the selected replica.
- On writing the object: the transaction accesses some majority of the replicas, determines the highest version number in the majority, generates a new version number by increasing the highest version number, and updates all replicas in the accessed majority with this new version number.

The update majority problem consists for a set of replica managers to agree on the outcome of a transaction. A proper algorithm for solving the update majority problem has the following three properties: (1) *uniform agreement*: no two failure-free replica managers decide differently, (2) *non-blocking*: every failure-free replica manager that starts the protocol eventually decides, and *uniform validity*: every failure-free replica manager decides identically.

4. CONCLUSIONS

This paper has described a practical application of the backward error recovery theory. A detailed failure model of the target system has been presented. We have shown how the developed failure model can be used in selecting the appropriate recovery methods for the different system components. The reasons behind dividing the system structure into SIMD and MIMD parts have been explained. Recovery techniques for both parts have been selected and briefly described. Finally, an implementation of a stable storage was proposed. The chosen methods take into account that modification of the present hardware is unreasonable, and therefore are primarily software-based solutions.

Future work in the presented topic will mostly be related to the physical evaluation of the abstract model. We need to collect practical experiences and measurement data about the real failure behaviour of the various system devices. Based on the collected data the model will be refined, and the assumptions will be validated to select the most suited rollback technique from the given alternatives. We also need to adapt the algorithms to the particularities of APEmille, and minimize the intrusion of the recovery subsystem by locating the possible performance bottlenecks.

5. REFERENCES

Aglietti, F., A. Bartolini, C. Battista and S. Cabasino (1997). The teraflop parallel computer APEmille. *Lecture Notes in Computer Science* **1225**, 991–998.

Aglietti, F. and et al. (The APEmille Collaboration) (1997). An overview of the APEmille parallel computer. *Nucl. Instr. and Meth. in Phys. Res.* **A 389**, 56–58.

Alvisi, L. and K. Marzullo (1995). Message logging: pessimistic, optimistic, and causal. In: *Proc. of the 15th Int. Conf. on Distributed Computing Systems.* IEEE Comput. Soc. Press. Los Alamitos, CA, USA. pp. 229–36.

Banatre, M., G. Muller and J.P. Banatre (1988). Ensuring data security and integrity with a fast stable storage. In: *Proc. IEEE Intl. Conf. on Data Eng.*. Los Angeles, CA. pp. 285–293.

Bartha, Tamás (1998). Rollback recovery in distributed systems. Technical Report IEI:B4-12-06-98. IEI CNR.

Bartha, Tamás (2000). A proposal for the recovery subsystem of the APEmille parallel computer. Technical report. IEI CNR. accepted for publication.

Chandy, K.M. and L. Lamport (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems* **3**(1), 63–75.

Chessa, S., B. Sallay and P. Maestrini (1999). Diagnostic model and diagnosis algorithm of a simd computer. In: *Proc. of the Third European Dep. Comp. Conf. (EDCC-3)*. Lecture Notes in Computer Science. Springer Verlag. pp. 283–300.

Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Communications of the ACM* **34**(2), 56–78.

Gibson, Garth Alan (1991). Redundant Disk Arrays: Reliable, Parallel Secondary Storage. PhD thesis. University of California at Berkeley. also available from MIT Press, 1992.

Guerraoui, R., R. Oliveira and A. Schiper (1996). Atomic updates of replicated data. In: *European Depend. Comput. Conf. (EDCC-2)*. LNCS. pp. 365–381.

Johnson, D.B. and W. Zwaenepoel (1987). Sender-based message logging. In: *17th Int. Symp. on Fault-Tolerant Computing (FTCS-17)*. IEEE Comput. Soc. Press. Washington DC, USA. pp. 14–19.

Lampson, B.W. (1981). Atomic transactions. In: *Distributed Systems–Architecture and Implementation*. Vol. 105 of *Lecture Notes in Computer Science*. pp. 246–265. Springer-Verlag. New York, NY.

Lampson, B.W., Paul, M. and Siegert, H.J., Eds.) (1981). *Distributed Systems - Architecture and Implementation. An Advanced Course.* Vol. 105 of *Lecture Notes in Computer Science*. Springer-Verlag. New York, NY.

Maestrini, P. and P. Santi (1995). Self diagnosis of processor arrays using a comparison model. In: *Symposium on Reliable Distributed Systems (SRDS'95)*. IEEE Computer Society Press. Los Alamitos, Ca., USA. pp. 218–228.

Pâris, J.-F. (1994). The management of replicated data. *Lecture Notes in Computer Science* **774**, 305–311.

Pâris, J.-F. and D.D.E. Long (1990). The performance of available copy protocols for the management of replicated data. *Performance Evaluation* **11**, 9–30.

Plank, J.S. (1993). Efficient checkpointing on MIMD architectures. Technical Report TR-406-93. Princeton University, Computer Science Department.