Facing up to the Inevitable: Intelligent Error Recovery in Massively Parallel Processing in Memory Architectures

James Kramer and Matthias Scheutz
Artificial Intelligence and Robotics Laboratory
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
Email: {jkramer3,mscheutz}@cse.nd.edu
Phone: (574) 631-{8380,0353}
Fax: (574) 631-9260

Jay Brockman and Peter Kogge
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
Email: {jbb,kogge}@cse.nd.edu
Phone: (574) 631-{8810,6763}
Fax: (574) 631-9260

Abstract*— Massively parallel "Processing-In-Memory" (PIM) architectures have been shown to yield increases in performance due to their "memory-centric" nature. However, as PIM is still a developing technology, advanced issues such as error detection *and* failure recovery *have not yet been addressed. We describe the application of concepts found in our multi-agent system,* `ADE`*, to PIM, incorporating its mechansims for automatic and intelligent error detection, failure recovery, and dynamic system reconfiguration in the PIM architecture, enhancing architecture robustness.

***Index Terms*— Fault tolerance, System recovery, Intelligent systems, Parallel architectures, Distributed computing**

I. Introduction

"We consider errors by people, software, and hardware to be facts, not problems that we must solve, and fast recovery is how we cope with these inevitable errors" [1]. This guiding principle of *recovery oriented computing* (ROC) succinctly summarizes some of the critical challenges that future massively parallel computing architectures need to address. The problem of faulty computations (regardless of whether they are due to errors committed by people, software crashes, or hardware faults) is exacerbated in the context of long-term, autonomous operations of massively parallel applications (e.g., the computing infrastructure of future space ships). Hence, while the first aim in designing, implementing, and operating massively parallel applications is to avoid or reduce the potential for errors, the second, by practical necessity, has to focus on how to deal with contingencies and recovery from failure.

"Recovery", in this context, must satsify various requirements. For example, *error detection* has to occur prior to recovery, which, in turn, requires *system monitoring* and possibly *introspection*. Moreover, the process of continuing or recovering interrupted, crashed, or otherwise disturbed computations necessarily involves mechanisms for *saving state* and determining the *dynamic dependency* of parallel computational processes. Finally, both software and hardware faults may necessitate the *dynamic (re-)configuration* of the software infrastructure of a computational system (e.g., relocating processes in memory, reassigning them to computational units, restarting them based on saved state, etc.). Such system reconfigurations need to be fast, efficient, and sensitive to the computational and communication bandwidth requirements of the processes involved.

We believe that the intelligent distribution, monitoring, and (re-)configuration mechanisms that have been investigated over the last several years in multi-agent system (MAS) research (e.g., [2], [3]), can provide effective mechanisms for implementing a "recovery oriented computing" infrastructure in massively parallel computing systems. Specifically, we propose applying the techniques for system monitoring, introspection, error detection, and recovery of a specific MAS developed in our lab, called `ADE` [4], [5], to the infrastructure of massively parallel "processing-in-memory" (PIM) architectures [6], [7], [8], where computational threads can be run locally on a simple processor within a memory chip (without the need of a main processor). We will describe how "agents" (in `ADE`) can be deployed throughout memory to form a tight network of memory clusters, organized around memory nodes, which are monitored both for software and hardware faults. Error recovery, then, involves fault detection within a cluster (or on a chip) and subsequent relocation of affected processes to other clusters such that bandwidth requirements are still met. To achieve this reconfiguration, special `ADE` components reason in real-time about the state of their clusters and the rest of the system to determine the best configuration given the dynamic state changes (e.g., faulty memory components, new memory that has become available, software components that crashed, etc.).

This paper proceeds as follows: Section II contains background material and a more detailed problem de-

scription. Section III provides overviews of both PIM and ADE, then discusses how the two can be brought together. A brief overview of related work in fault-tolerant and reconfigurable hardware is given in Section IV, followed by a discussion (including some experimental results) in Section V and a conclusion in Section VI.

II. Requirements for Error Detection and Recovery in Massively Parallel PIM Architectures

Processing-In-Memory (PIM) technology is being developed to address an increasingly critical problem in high-performance computing, namely the performance limitations imposed by the "memory wall" associated with transferring data between the processor and memory [9]. The problem gets worse in subsequent generations of technology, as applications continue to become more memory-intensive. PIM architectures address this problem by taking a *memory-centric* approach, moving computation directly into the memory system. In doing so, PIM architectures have been found to have a significant positive impact on system performance, while also making chip design more managable via the tiling of relatively small, regular logic structures [10].

While PIM architectures can significantly improve the performance of computing systems by offloading processes from main processors into memory, they are still, like all other massively parallel architectures, subject to performance bottlenecks (e.g., processes that communicate with each other are not located on the same or in adjecent memory chips, thus requiring higher network traffic). Moreover, in the long-term, computational components will eventually fail (for reasons including software bugs, hardware faults, network unavailability, etc.). The problem of single component failures is exacerbated when considering multiple component failures, which can potentially occur at multiple sites at the same time, and may, in the worst case, lead to large-scale system failures. Especially in massively parallel system with millions of parallel processes, individual component failures will be difficult, if not impossible, to detect (e.g., components might give the appearance of working properly, or might not respond for an extended period of time, or might crash only occasionally under difficult-to-trace conditions). Without automatic monitoring of component execution, determining the circumstances under which they fail is problematic. Moreover, without automatic recovery mechanisms, it will be impossible to maintain system integrity and resume individual computations (e.g., from the last saved state prior to the time of fault occurrance) without risking large-scale, catastrophic system failure.

Various techniques have been proposed to detect and alleviate errors and failures in massively parallel computing systems, such as *redundant processing* (i.e., replication), *concurrent error detection*, *watchdog processes*, *checkpointing*, etc. [11], [12], [13], [14]. A price is paid in each case, characterized in terms of *overhead* (e.g., additional time, memory space, hardware, etc.). For instance, replication requires (at least) multiplication of computational hardware (for simultaneous execution) or processing time (for serial execution). Alternately, checkpointing requires mechanisms that periodically save processing state, with the associated costs of additional complexity, requisite memory, and interruption of the process. The frequency and efficiency of the checkpointing mechanism will determine how much space is required, how much data is lost during recovery, and what needs to be re-computed afterwards.

In every case, at a minimum, some form of *monitoring* is required to identify when and where an error or failure has occurred, in addition to a *policy* that dictates the steps of the recovery process (e.g., information about restarting computations, or continuing them if saved state is available). Call this *minimum recovery oriented computing* (MROC). Given the dimensions of massively parallel systems, MROC itself requires distribution, as otherwise it would create an intolerable performance bottleneck. Hence, at any given time, multiple MROC mechanisms will be at work simultaneously at different places in the system, each in charge of some local environment and some remote MROC components, which also need recovery if their underlying hardware infrastructure fails.

Due to their distributed nature, MROC components require additional synchronization and communication mechansims that allow them to both notify remote components of local failures and obtain status and recovery information from the remote components. Recovery in such a system is then a highly distributed process that can ultimately lead to a systematic restart of the whole system (from a saved state) if all local recovery efforts fail. Ideally, this process should not only reliably recover from failures, but also be sensitive to the processing requirements of the recovered components (e.g., memory availability, communication bandwidth, etc.). Hence, it is desirable for MROC to be able to allocate resources in a way that attains the best (or near-best) performance for the recovered components, given the dynamic system contraints at the time of recovery (both in terms of hardware and software).

III. Combining PIM and ADE

Because PIM is a developing technology, systematic error detection and error recovery of hardware and software components has not yet been addressed. We propose adapting our existing multi-agent ADE system, designed for fault-tolerant and real-time distributed computing in heterogeneous computing infrastructures, to PIM architectures. Specifically, we modify "heavy-

weight" ADE components (intended to run as computational processes on server hosts) to run as "lightweight" threads on PIM memory nodes, while retaining their monitoring and introspection mechanisms as well as the reasoning mechanisms incorporated into the special ADE infrastructure components responsible for dynamic system (re-)configuration capabilities. Critically, by implementing ADENodes as PIM threads that run in memory without using any computation time of the main CPUs, the proposed adaptation will overcome the performance bottleneck imposed by systematic large-scale ROC components, thus combining the advantages of processing in memory with the intelligent system monitoring, introspection, and reconfiguration mechansims of ADEthat give rise to a highly robust, reliable and efficient autonomic computing infrastructure.

We start with a brief overview of the PIM architecture and the ADE system, and then present the details of the adaptation, which can, in addition to performing error detection and recovery, take performance needs of recovered components into account.

A. An Overview of PIM

Processing-In-Memory, or PIM, [6], [7], [8] involves coupling large amounts of logic and memory on a single VLSI chip to greatly reduce latency. Rather than denoting a specific implementation, PIM is a collection of methods and technologies that cover all aspects of pushing computation into the memory system, including programming and execution models, microarchitectural organization, and physical design and layout. However, in this case, we use "PIM" to refer to the PIM-Lite [15], [16], [10] reference implementation, in addition to the related HTMT [17], [18] and DIVA [7] projects.

Using PIM, accesses from logic to memory no longer have to go through the memory hierarchy; rather, computation "follows the data" to memory, yielding up to an order of magnitude reduction in effective latency, directly addressing the "memory wall" [9] of computer performance. Two derivative benefits are immediately gained: (1) a reduction of total silicon area, as driver/receivers, off-chip pads, and long range clocking logic are largely eliminated and (2) a reduction of power consumption due to removal of logic circuitry.

The PIM-Lite ISA was designed from the outset to support multiple threads, minimizing the cost of moving thread state by keeping most of it in memory at all times, while the microarchitecture supports simultaneous multi-thread processing. In the PIM model, the state information of a thread is kept in memory as a *data frame*, a region of contiguous memory locations within a single node. The execution state of a PIM-Lite thread is completely described by a *continuation* consisting of two pointer values: a frame pointer, FP, which points to the starting location of the data frame, and an instruction pointer, IP, that points to the current instruction. Threads in a PIM node are kept in a *thread pool*, responsible for scheduling their execution; the compactness of the continuation representation allows context switches to take place in single clock cycle. Threads that operate on a common frame are synchronized with a counting semaphore in the frame slot.

When a PIM thread needs to access data that is not on the current node, there are two choices: either bring the data to the thread, or move the thread to the data. Given the small thread state and the likelihood of future references to data on a node, the latter option is especially attractive. The determination as to whether a given virtual address is on the current node or not (i.e., the *locality*) is established by checking the local page table; if an address is not local, PIM supplies *conditional memory transfer* instructions. Remote data is accessed via a *parcel*, a very lightweight message passing mechanism for transferring frame and continuation data between nodes.

B. An Overview of ADE

ADE [4], [5] is a framework for agent development that provides robust and reliable middleware services for the distribution of complex agent architectures across many hosts. In addition to facilitating parallel operation of components, it includes mechanisms for monitoring, error detection, and recovery services that are operational at application run-time to provide a fault-tolerant computational environment.

The basic component in ADE is an ADENode, which is comprised of one or more computational processes (potentially with multiple threads) that serve requests. Accessing the services provided by an ADENode is accomplished via inter-process communication through some message passing interface. A "supervisor", a special type of ADENode, monitors the status of ADENodes, keeping track of the response of nodes that "register" with it. The supervisor provides the backbone of an ADE system; all nodes must register with a supervisor to gain access to the monitoring and recovery mechanisms. A set of ADE components may contain multiple supervisors that mutually register with one another (to form a connected graph of sub-graphs), providing both mutual redundancy and the means to maintain distributed knowledge about the system.

All the components of an implemented architecture (that is, all ADENodes and supervisors) maintain a communication link during system operation to provide status information. At a bare minimum, this consists of a periodic *heartbeat* signal that indicates a component is still functioning (although messages may contain arbitrary data, such as an error code or other information about internal component operation). Heartbeats form the basis for system-wide component failure detection, used to detect and react to failures resulting from hardware and software problems (e.g., programs not

responding, malfunctioning computational units, etc.). In the simplest case, non-reception of a heartbeat is indicative of system disruption (i.e., inaccessibility of an ADENode), causing total component restart. Slightly more complicated is an error code that indicates faulty component operation, which may require a specially defined procedure for fault correction.

As mentioned above, supervisors are a special form of ADENode that are able to mutally register with one another. Each retains knowledge about registered ADENodes and is able to transfer that information to other supervisors, thereby creating a recursive set of distributed monitoring mechanisms. Internally, in addition to mechanisms for testing target node availability, these supervisors consist of high-speed reasoning engines that allow them to quickly determine potential causes for ADENode faults (e.g., software bugs or non-repsonsive nodes) and find the best policy for responding to the fault(s) (e.g., restart, relocation, reconfiguration, etc.). ADE has been experimentally validated, demonstrating its utility as a robust, fault-tolerant infrastructure for distributed applications (as described in Section V).

C. Implementing Light-weight ADENodes in PIM Threads

As described above, all components in an ADE system maintain *heartbeat* signals; a missing heartbeat indicates a failed component. When an ADENode receives no acknowledgment of a sent heartbeat, the heartbeat is suspended until a replacement supervisor "reconnects", at which point the heartbeat is redirected to the new supervisor. When a supervisor does not receive a heartbeat in some allotted time, the sending ADENode is presumed failed and restarted (either on the same host, or elsewhere if the original host is unavailable). Hence, two basic mechanisms are necessary to translate ADE's failure recovery to PIM architectures: communication of *heartbeat* messages and the supervisor's ability to probe for host availability.

Given PIM's lightweight and multi-threaded computational model, in which context switching takes a single clock cycle, heartbeats can be implemented as individual threads that send parcel messages to a supervisor, while a supervisor probe can be a thread that is executed remotely. As each ADENode is a single thread, it is feasible to assign one ADENode to each PIM node, assigning subsets of ADENodes to supervisors (which themselves are special kinds of ADENodes with only infrastructure functionality). Each non-supervisor ADENode automatically connects on startup to its assigned supervisor and starts its heartbeat signal. An ADENode that receives no response to a heartbeat behaves just as described above, as does a supervisor (substituting "PIM node" for "host").

In addition to sending regular heartbeat signals, each ADENode keeps track of threads executing on the local PIM node. To do so, each thread is required to notify the resident ADENode on startup, relocation, and shutdown; the ADENode confirms threads' operation by examining the contents of the local thread pool. This notification process allows an ADENode to determine when threads exit abnormally (e.g., due to errors), at which point the ADENode forwards this information to its supervisor, which will then take appropriate action (e.g., by asking the ADENode to restart the thread).[1] If restart is not possible on the PIM node (e.g., due to resource limitations such as newly started threads or hardware failures), the supervisor can either relocate the thread to another PIM node that it supervises, or request a transfer of the thread to the ADENode pool of another supervisor (via a broadcast message as part of the supervisor heartbeats).

D. Improving Performance

The description of failure recovery given above is suitable for any set of high-level OS processes (or heavy-weight threads) that involve light-weight (i.e., PIM) threads across PIM nodes. While it is possible to assign arbitrary locations to all light-weight threads in a heavy-weight process/thread, such random assignment could significantly impact performance (as PIM threads that need to communicate, but do not reside on the same PIM node need to be transferred through the PIM network). Such *locality* considerations can be addressed with ADE techniques by defining an appropriate set of dynamically updated *facts* that can be *introspected upon* and *reasoned about* with an appropriate *policy*.

Specifically, the facts used to address locality include: (1) a unique identifer (ID) for each PIM node, (2) the threads currently executing on a PIM node, (3) information about the system topology, and (4) the "relatedness" of threads. If not explicitly specified, (1) can be derived from the PIM node's system memory location, while the data for (2) is contained in an ADENode's heartbeat (for efficiency, the heartbeat transmits only changes in the thread list or an "OK" code if no threads have started or terminated). We assume that (3) is made available via some simple data structure that represents the system (e.g., the number of threads available per PIM node, the interconnections among PIM nodes, the network latency, etc.) that may reside in a remote memory location; for redundancy purposes, it can also be replicated at multiple known memory locations. To specify (4), threads are considered "related" when they either perform calculations on shared memory or when one depends on the other, as determined by memory access pattern.

To obtain the best locality, threads should not only be located near the memory data they access, but related

[1] We are currently investigating the possibility of a mixed "heavy-weight/light-weight" ADE system, in which the main CPUs run heavy-weight ADE components, which are directly connected to light-weight supervisors. That way supervisors can initiate more complex recovery actions at the system level that can use any of the main heavy-weight processors.

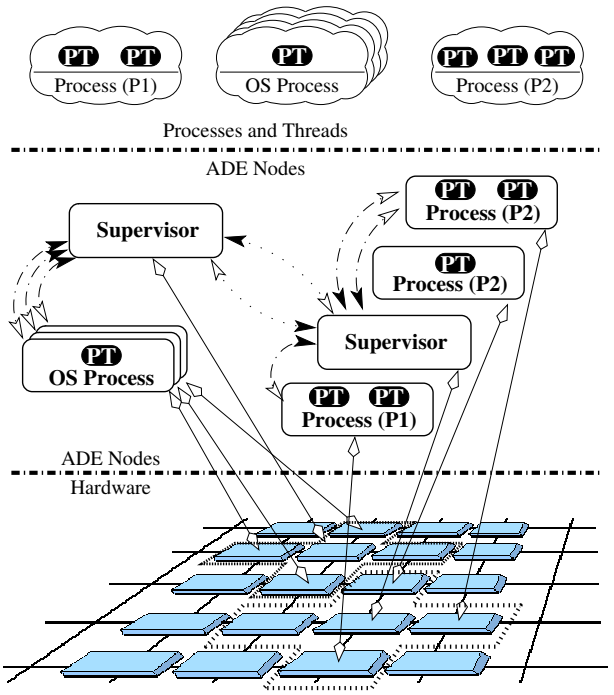

Fig. 1. An example PIM/ADE configuration using a grid topology. **Top:** the processes and related PIM threads (PT). **Middle:** the ADENodes that implement the processes and threads shown on the top level. **Bottom:** twenty PIM nodes in a grid topology; arrows indicate an ADENode's PIM node location, while a *neighborhood* is shown by a broken line surrounding five PIM nodes (there are two in the figure, each with a supervisor at its center).

threads should be located near one another as a group, where "near" is defined exactly as occupying locations that minimize latency. For example, a system topology may be a two-dimensional grid (as shown on the bottom of Fig. 1), such that nodes located next to one another have the lowest latency. Note that while minimzing latency may correspond to physical distance (as in the figure), it is actually a function of the particular topology (e.g., a torus or hypercube would exhibit different latencies).

Configuration–that is, assigning threads to locations–is handled by a set of supervisors, each of which is located in the center of a *neighborhood*, defined as the set of PIM nodes adjacent to and including the supervisor's PIM node. As described earlier, supervisors contain functions for starting, stopping, and relocating threads. In addition, they have functions for communicating with other supervisors, allowing them to obtain information about remote neighborhoods. They also contain the facts (mentioned above) about both their local neighborhood and the system topology in a simple list form, as well as rules that form a policy.

Let $N(p)$ be the neighborhood of PIM node p, t be the number of a PIM node's available thread slots, and T be a given set of interacting threads. The best configuration k (i.e., the minimal number of PIM nodes on which to place the threads) is given by $k = |T|/t$ (in the worst case, where each PIM node has only a single available thread, $k = T$). To maintain maximal locality, we wish to preserve the relation $k \leq N(p)$ for each T, where the location of $N(p)$ coincides with the memory accessed by the threads in T.

There are multiple cases that have to be addressed. In the case of an individual thread failure, the supervisor's job is find a PIM node in its neighborhood on which to restart the thread. In the case of a PIM node failure, the supervisor is not only responsible for restarting all the failed threads, but also locating them properly. This may entail relocating any or all threads to another node within the local neighborhood, or potentially sending them to another neighborhood, transferring responsibility to another supervisor. Finally, there is also the case where a supervisor itself or the node on which it is located fails, handled by using redundant supervisors located on a different PIM node. These supervisors are not actively involved in "neighborhood maintenance", but communicate only with the primary supervisor. The specifics of all of the above are dictated by the policy rules in place for a given supervisor; not only can each policy be different, but policies may be adapted during system operation.

IV. Related Work

Error detection and fault-tolerance has been addressed in various ways in the literature. One error detection technique is the use of *watchdog processors* [11], which concurrently monitor calculations to detect errors. In fact, the supervisors described above perform a similar function, following the same two-phase process: *setup*, where the watchdog is given information about the calculations to check and *checking*, where the information is collected concurrently. Supervisor components are more flexible, however; they do not require special hardware (as they are implemented in threads that can execute on any PIM node), while the rule-based policy used for checking can be adapted or replaced at run-time (simply by changing the rule representations). Moreover, the policy can include a number of detection methods, as well as a variety of recovery techniques, that are also adaptable and can be replaced at run-time. Similarly, supervisory components can be used to facilitate or implement various *concurrent error detection* (CED) schemes, such as those found in [13] (e.g., duplication of processing, parity prediction, and unidirectional codes).

Various methods for fault recovery have also been proposed. For instance, Lach *et al.* [12] give algorithms for efficient run-time fault recovery on FPGA architectures. In the discussion, *error detection*, *locality*, and *diagnosis* are assumed; we have described concrete detection and locality mechanisms above, while diagnosis in PIM and ADE is performed at a node-granularity (which,

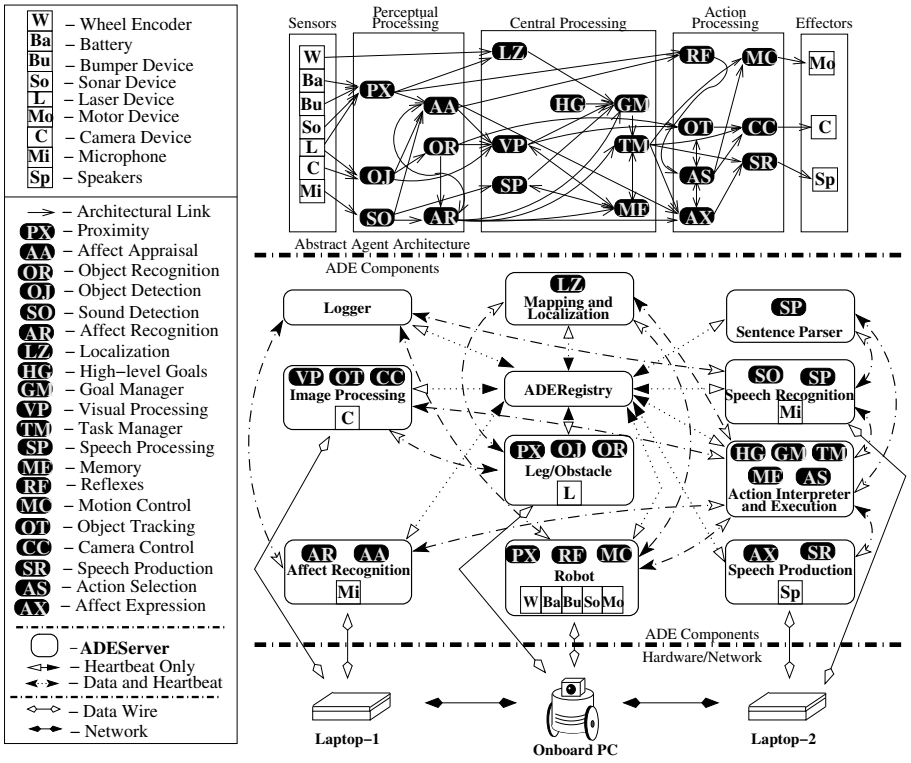

Fig. 2. The ADE configuration used in the experiments described in Section V. **Top:** the abstract agent architecture; data flow goes left to right, starting with sensory input, going through various stages of processing, ending with effector output. **Middle:** one implementation of functional components as ADE components, where arrows within the level indicate communication channels; note that the ADERegistry component is the "heavy-weight" version of the supervisor for the PIM architecture. **Bottom:** the hardware (two laptops and a robot with an on-board PC that are connected via an internal network) on which the ADE components execute, located directly above the host on which they reside, where arrows that cross levels indicate hardware connections.

in the simplest case, corresponds to Lach's definition and consists of treating a node as permanently failed). Unlike these algorithms, the proposed PIM and ADE setup is free from "providing multiple configurations"; rather, dynamic configurations can be automatically and intelligently determined.

Another example of a FPGA fault-recovery technique is the *roving Self-Testing ARea approach* (STAR) approach [14]. The PIM and ADE combination is similar in that it also "integrates on-line test, diagnosis, and fault-tolerance in a unified fromework". However, testing in our described setup is done on active PIM nodes with no interruption of system operation. Furthermore, the PIM and ADE system has no need for pre-defined configurations, but can determine a configuration dynamically, thus allowing for run-time re-configurations.

V. Experimental Results

To evaluate the utility of ADE's infrastructural error detection and recovery mechanisms, which rely on introspection and reasoning about infrastructure configurations, we conducted proof-of-concept experiments in a "heavy-weight" ADE environment implemented on an assistive robot that has to interact with humans using natural language in a joint human-robot task (depicted in Fig. 2; for details, see [19]). Experiments consider simulated *catastrophic hardware failure* of one host in a classical networked configuration; in particular, the failure of an entire computational unit (i.e., host computer) and, therefore, the software components executing on that unit, at run-time, in a time sensitive, dynamic environment. The *time-to-task-completion* was recorded for four separate cases: (1) no failure occurs, providing a best case scenario, (2) failure occurs with no active recovery mechanisms, providing a worst case scenario, (3) failure occurs with recovery mechanisms active and redundant components already executing, and (4) failure occurs with recovery mechanisms but without redundant components. Results are given in Table I.

TABLE I
AVERAGE TIME TO COMPLETION (IN SECONDS) OVER TEN RUNS FOR COMPUTATIONAL HOST FAILURE.

Category	Avg. Time (s)	Std. Dev.
No failure	75.47	8.0
Recovery w/ redundancy	87.24	14.0
Recovery w/o redundancy	103.19	26.0
No recovery	∞	n/a

In addition to the quantitative measure of failure recovery provided by the above experiments, we have

also demonstrated (inadvertently) the extent to which the underlying ADE system can detect component crashes and recover from them in a qualitative way. In particular, ADE was executing on computers in student cluster environments at the University of Notre Dame; the only way to stop ADE's recovery process was to take whole clusters offline by shutting down the switches that connected them to the outside world. Given enough computational units, it will be practically impossible for failures to bring down the whole ADE system.

It is important to note that the data reported from the above proof-of-concept experiments of ADE's failure recovery mechanisms reflect the use of "heavy-weight" components in a multi-host environment that relies on TCP/IP networking. Failure detection in that environment takes approximately 8 seconds, while component recovery or restart requires an additional, component-dependent time. When incorporated into PIM architectures, the corresponding error detection and recovery of light-weight ADENodes will occur in times that are several orders of magnitude faster. Since only a few prototypes of PIM chips are currently available that are inadequate for performance testing, we expect to obtain fine-grained performance data of error detection and recovery in the near future via the SALT simulator. SALT is a purpose-written simulator that is intended to provide a software testbed for threaded applications in PIM architectures that is currently in the final stages of its development, designed to provide detailed performance data for threaded applications (see [10] for more on the simulation models). Hence, once completed, SALT will allow us to run and test the adapted ADE system and collect quantitative data about the functionality of ADE and the various overheads (e.g., in terms of memory and PIM threads).

VI. Conclusion

We have described how concepts established and implemented in the context of a distributed multi-agent system like ADE can be leveraged and applied to the hardware level of massively parallel computing infrastructures like PIM systems in order to implement *recovery oriented computing*. In addition to providing preliminary experiments that confirm the viability of the ADE approach at the classical network level, the method of PIM implemention has been laid out and is awaiting completion of the SALT simulator to begin initial testing. We expect this combination of ADE and PIM to exceed the performance achieved by other error detection and recovery methods, while putting little to no strain on heavy-weight CPUs due to its complete embedding in PIM memory. By allowing dynamic reconfigurations, the proposed mechanism will provide the basis for achieving the introspective system awareness required for the long-term, robust, and autonomic operation of massively parallel computing systems.

References

[1] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies," UC Berkeley Computer Science, Tech. Rep. UCB//CSD-02-1175, 2002.

[2] F. Bellifemine, A. Poggi, and G. Rimassa, "JADE - a FIPA-compliant agent framework," in *Proc. of the 4th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, 1999, pp. 97–108.

[3] K. Sycara, M. Paolucci, M. V. Velsen, and J. Giampapa, "The RETSINA MAS infrastructure," *Autonomous Agents and Multi-Agent Systems*, vol. 7, no. 1, pp. 29–48, 2003.

[4] V. Andronache and M. Scheutz, "Integrating theory and practice: The agent architecture framework APOC and its development environment ADE," in *Proc. of Autonomous Agents and Multi-Agent Systems*, 2004, pp. 1014–1021.

[5] M. Scheutz, "ADE - steps towards a distributed development and runtime environment for complex robotic agent architectures," *Applied Artificial Intelligence*, vol. 20, no. 4-5, 2006.

[6] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovíc, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable processors in the billion-transistor era: IRAM," *IEEE Computer*, vol. 30, no. 9, pp. 75–78, 1997.

[7] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Drapper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *ACM International Conference on Supercomputing (SC'99)*, 1999.

[8] G. Kirsch, "Active memory device delivers massive parallelism," in *Microprocessor Forum*, San Jose, CA, 2002.

[9] A. Saulsbury, F. Pong, and A. Nowatzyk, "Missing the memory wall: The case for processor/memory integration," in *Proc. of the 23rd International Symposium on Computer Architecture*, 1996, pp. 90–101.

[10] S. Thoziyoor, S. Kuntz, J. Brockman, and P. Kogge, "Cost/performance analysis of a multithreaded PIM architecture," 2005, IEEE Transactions on VLSI, under review.

[11] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors-a survey," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 160–174, 1988.

[12] J. Lach, W. Mangione-Smith, and M. Potkonjak, "Algorithms for efficient runtime fault recovery on diverse FPGA architectures," in *International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '99)*, 1999, pp. 386–394.

[13] S. Mitra and E. McCluskey, "Which concurrent error detection scheme to choose?" in *Proc. of International Test Conference*, 2000, pp. 985–994.

[14] J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici, "Dynamic fault tolerance in FPGAs via partial reconfiguration," in *Proc. 8th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM '00)*, 2000, pp. 165–174.

[15] J. Brockman, P. Kogge, S. Thoziyoor, and E. Kang, "PIM lite: On the road towards relentless multi-threading in massively parallel systems," University of Notre Dame, Tech. Rep. TR-03-01, 2003.

[16] S. Thoziyoor, J. Brockman, and D. Rinzler, "PIM lite: A multithreaded processor-in-memory prototype," in *GLSVSLI '05: Proc. of the 15th ACM Great Lakes Symposium on VLSI*, 2005, pp. 64–69.

[17] J. Brockman, P. Kogge, V. Freeh, S. Kuntz, and T. Sterling, "Microservers: A new memory semantics for massively parallel computing," in *Proc. of the 1999 International Conference on Supercomputing*, 1999, pp. 454–463.

[18] T. Sterling and L. Bergman, "A design analysis of a hybrid technology multithreaded architecture for petaflops scale computation," in *Proc. of the 1999 International Conference on Supercomputing*, 1999, pp. 286–293.

[19] M. Scheutz, P. Schermerhorn, J. Kramer, and C. Middendorff, "The utility of affect expression in natural language interactions in joint human-robot tasks," in *ACM Conference on Human-Robot Interaction (HRI2006)*, 2006.