A Fault Detection and Recovery Architecture for a Teradevice Dataflow System

Sebastian Weis*, Arne Garbade*, Julian Wolf*, Bernhard Fechner*,
Avi Mendelson†, Roberto Giorgi‡, Theo Ungerer*

*University of Augsburg
Universitaetsstr. 6a
86159 Augsburg, Germany
{surname}@informatik.uni-augsburg.de

†Microsoft R&D Israel
13 Shenkar
Herzliya, Israel
avim@microsoft.com

‡University of Siena
Via Roma 56
53100 Siena, Italy
giorgi@dii.unisi.it

ABSTRACT

Future computing systems (Teradevices) probably contain more than 1000 cores on a single die. To exploit this parallelism, threaded dataflow execution models are promising, since they provide side-effect free execution and reduced synchronization overhead. But the terascale transistor integration of such chips make them orders of magnitude more vulnerable to voltage fluctuation, radiation, and process variations. This means reliability techniques have to be an essential part of such future systems, too.

In this paper, we conceptualize a fault tolerant architecture for a scalable threaded dataflow system. We provide methods to detect permanent, intermittent, and transient faults during the execution. Furthermore, we propose a recovery technique for dataflow threads.

1. INTRODUCTION

Nowadays, the number of transistors still increases, but no longer with significant frequency enhancements and the cost of extra power and power density. These facts open the doors for new highly scalable computing systems with probably more than 1000 cores (Teradevices) and an increasing need for exploiting such large amount of parallelism.

The International Technology Roadmap for Semiconductors [1] prognoses that a shrinking feature size and decreasing supply voltage leads to increasing failure rates of up to 400% [30]. Also, complexity and costs for testing and verification of devices will increase. With the ongoing decrease of the transistor size, the probability of physical flaws on the chip, induced by voltage fluctuation, cosmic rays, thermal changes, or variability in the manufacturing process will further raise [30], making faults in present multi-core and future many-core systems unavoidable.

While in mission critical systems fault-tolerance has always been essential, the architecture of a general purpose processor is strongly influenced by economical constraints. This requires fault-tolerance techniques able to scale with the number of cores and the increasing failure probability on a chip in conjunction with a reasonable architectural effort [5].

Threaded dataflow [9, 12, 31, 34] is known to overcome the limitations of the traditional von Neumann architecture by exploring the maximum thread-level parallelism of the hardware and reducing the synchronization overhead. It is not only a promising candidate to exploit parallelism in future Teradevices, but can also serve as a basis for a fault-resilient parallel architecture. The pure single-assignment and side-effect free semantics of dataflow threads provide an advantage for recovery and double execution techniques compared to state-of-the-art von Neumann threads.

The contribution of this paper is the presentation of a fault-tolerant architecture for a parallel and hierarchically threaded dataflow system. In detail, we provide techniques like redundant execution, control flow checking, and checkpointing to cope with transient, intermittent, and permanent faults on different architectural levels. Moreover, we propose a lean recovery mechanism on thread-level.

The paper is organized as follows: in Section 2 we present related work. For the convenience of the reader, we provide different sections related to different fault detection mechanisms. Section 3 describes the underlying dataflow architecture. Based on this, we describe the fault detection extensions in Section 4. The mechanism to recover from faults is presented in Section 5, followed by a conclusion in Section 6.

2. RELATED WORK

2.1 Fault Tolerance in Macro Dataflow Architectures

The benefits of a side effect free execution model for fault tolerance have already been studied in the context of different macro dataflow architectures. Nguyen et al. [19] proposed a fault tolerance scheme for a wide-area parallel system, considering a macro dataflow architecture built on top of a wide-area distributed system. This differs from our architecture as we target a single chip multiprocessor system with hardware support for thread scheduling and fault tolerance.

Another technique by Jafar et al. [14] exploits the macro dataflow execution model of KAAPI [8] for a checkpoint/-recovery model. KAAPI uses a C++ library on commodity chip multiprocessor clusters that exposes a dataflow programming model. Since KAAPI is a software library the model has to cope with the overhead usually introduced with software fault tolerance techniques. Our work mainly focuses on hardware fault tolerance schemes.

2.2 Redundant execution

The redundant execution of threads can take various forms [6, 28]. We distinguish between spatial and temporal redundancy. In the case of spatial redundancy, duplicated instances of a thread are executed in parallel on separate hardware. In a temporal redundant system, the threads execute subsequently on the same hardware. Rotenberg [26] was the first who used an SMT-capable processor for temporal redundancy, which executes duplicated threads multithreaded on one processor.

Mukherjee et al. [18] presented a combination of spatial and temporal redundancy for an SMT-capable multi-core system. Here, different threads are executed multithreaded on one core and additionally the redundant threads are distributed to other cores.

Our redundant execution approach (see Section 4.6) combines both temporal and spatial redundancy on thread-level to detect permanent, intermittent, and transient faults.

2.3 Control flow checking

The detection of control flow errors has been an open research topic for more than two decades [16]. The developed approaches can be basically organized into three categories, according to the implemented check mechanisms: *Hardware-based* techniques [17, 21, 27, 33] extend a processor with an additional hardware check unit, while *software-based* techniques [3, 11, 20, 25] add redundant instructions to harden critical parts of an application. *Hybrid* techniques [4, 24] combine both hard- and software detection mechanisms. On the one hand, such a hybrid mechanism should reduce the high memory and execution time overhead of software approaches, on the other hand, the complexity compared to pure hardware techniques is reduced as only parts of the check mechanism must be implemented in hardware.

2.4 Rollback-recovery mechanism

Elonzahy et al. [7] divide rollback-recovery for message-passing systems into *checkpoint-based* and *log-based* mechanisms. Checkpointing depends on restoring a global system state, while log-based mechanisms combine checkpointing with logging of non-deterministic events.

Prvulovic et al. [23] and Sorin et al. [29] have both described global checkpointing techniques with logging for the rollback-recovery in a shared memory multiprocessor.

Since our execution model provides inherent checkpoints between dataflow threads (see Section 3.2), we use a local thread-restart mechanism without the need for restoring a global state or the logging of events.

3. OVERALL ARCHITECTURE

Our assumed dataflow architecture and execution model builds upon the Decoupled Threaded Architecture (version for clustered architectures DTA-C) originally described in [9]. DTA-C is designed to fully exploit the Thread-Level

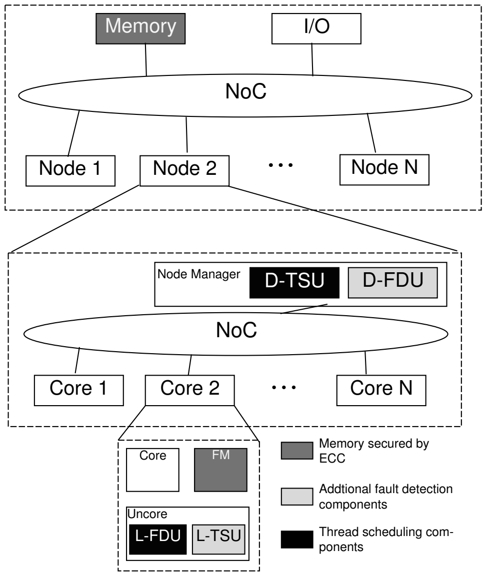

Figure 1: High-level architecture

Parallelism (TLP) provided by future parallel systems. It addresses scalability by a hierarchical structured execution model. Although it is based on DTA-C, our architecture differs in two points from the DTA-C approach. First, unlike in DTA-C, in this paper we do not imply a synchronization and execution pipeline but a standard x86-64 pipeline per core. Second, we incorporate x86-64 control flow instructions to support micro control-flow within a thread. This allows us to exploit data locality without replicating instructions or having to create new threads. For simplicity we call a dataflow thread with micro control flow support a *thread*.

3.1 Basic Architecture

We assume a tiled hardware architecture, where a tile is denoted as a node. As shown in Figure 1 each node is comprised of a certain number of cores and node management modules. In the following we describe these components in detail.

3.1.1 Core level

On the core level, the basic elements of our architecture are single cores containing an x86-64 pipeline (x86-64 ISA with dataflow extensions derived from [22]) along with a small unified L1-Cache. Each core includes special hardware extensions consisting of two modules:

- The *Local Thread Scheduling Unit* (L-TSU) is responsible for scheduling threads on its affiliated core and communicating with other L-TSUs or the node's D-TSU.
- The *Local Fault Detection Unit* (L-FDU) is responsible for the detection of faults and reliability management within a core.

Beside the L-TSU and L-FDU, each core stores the data of a running thread in the Frame Memory (FM). The Frame Memory is managed in a way that the data appears at the top level of the memory hierarchy (possibly all in the L1 cache [10]). This memory is filled with the thread's data (denoted as *thread frame*) before execution. Threads are not allowed to read from other thread frames. However, writes into disjoint locations are permitted to support communication between threads in order to provide the inputs for subsequent threads.

3.1.2 Node level

From the node level perspective we propose two additional hardware modules for management purposes. First, the *Distributed Thread Scheduling Unit* (D-TSU) coordinates the scheduling of the threads to cores within a node and communicates with other D-TSUs. Therefore, the D-TSU holds a table for bookkeeping the thread-to-core relations. Second, the *Distributed Fault Detection Unit* (D-FDU) is responsible for fault detection, performance monitoring, and reliability management within a node.

3.1.3 Communication

For the communication between nodes, we assume an interconnection network in style of a 2D-mesh. All communication from one node to another will be handled by the interconnection network. Furthermore, we consider memory controllers to access off-chip DRAM and I/O-controllers on node level. The controllers are connected to the interconnection network as well.

3.2 Execution model

A DTA-C program is partitioned in coarse-grained dataflow threads, where the execution of a thread consists of three phases. First, the pre-load phase loads data from the FM and stores it into the core registers. The second phase is the thread execution, where the thread executes without any memory access. The third phase is the post-store phase, where the results from the thread execution are written to the consuming thread frames.

Beside the frame, each thread has an assigned control structure called *continuation*. This structure stores control information about the thread, i.e. the pointer to the thread frame, the program counter, and the synchronization count (number of empty inputs). A thread will be scheduled for execution if and only if all inputs have been written to the thread's frame and therefore its synchronization count is zero.

Since in DTA-C prefetching can be very productively coupled with the scheduling of threads, accesses to FM usually have low latency and are not likely to suffer from page faults or cache misses. Generally, a core's pipeline is supposed to seldom stall in the case of FM accesses.

4. FAULT DETECTION EXTENSIONS

The central components of our fault detection approach are the already mentioned Fault Detection Units (FDUs). The Distributed FDU (D-FDU) is a lean hardware unit operating as an observer-controller on node level. As such, a D-FDU autonomously queries and gathers the health states of all cores within its node over the unreliable interconnect. In this context the D-FDU is supported by the L-FDUs (described in Section 4.2) located with each node's core. In addition, D-FDUs monitor each other in order to detect faults of other D-FDUs in other nodes. The D-FDU analyzes the gathered information and provides the thread scheduler on node level (D-TSU) with information about the state of the whole node and other D-FDUs.

4.1 Basic fault model

This subsection describes our underlying fault model. We assume non-systematic transient faults in the form of Single Event Upsets (SEUs), permanent, intermittent, and transient faults during operation in cores and interconnects.

SEUs and permanent faults are presumed to occur in one component at a time, since multiple bit faults at a time are extremely seldom. At this stage of development, a component can be a D-TSU, an L-TSU, a D-FDU, an L-FDU, a core, or a link.

On intra-node level, we assume

- permanent, intermittent, and transient faults within cores and L-FDUs and
- permanent and intermittent broken links between cores, L-TSUs, and L-FDUs.

On inter-node level our architecture has to cope with permanent, intermittent and transient faults of whole nodes or links between nodes, I/O, and memory.

We further hypothesize that all communication between cores, FDUs, TSUs, and memories, i.e. off-chip RAM and the FM within the nodes, is secured by error correcting codes (ECCs).

In the rest of this paper, we focus on the intra-node level.

4.2 L-FDU

The L-FDU is a small hardware unit implemented on each core to detect transient faults by extracting information from the Machine Check Architecture (MCA, see Section 4.4). Basically, the L-FDU has two tasks:

1. Reading out the fault detection registers of the monitored core, i.e. registers of the Machine Check Architecture or the Control Flow Checker, described in Section 4.4 and 4.5, respectively.
2. Periodic communication with the D-FDU by sending health messages of the core.

4.3 D-FDU

Concerning intra-node fault detection, the D-FDU detects node and link failures and informs the D-TSU about the faulty components, while the D-TSU is responsible for thread recovery and restart.

For the internal behavior of the D-FDU, we adopt an autonomic computing approach [15] organizing the operation principle into the four consecutive steps: Monitoring, Analyzing, Planning, and Executing (see Figure 2). This MAPE cycle operates on a set of managed elements, comprising intra-node (cores and D-TSU) and inter-node elements (other D-FDUs) in other nodes [32].

D-FDUs detect faults and proactively maintain the operability of the node they monitor, for example by dynamically performing clock and voltage scaling while monitoring the cores' error rates, temperatures, and utilization. In this context proactive means the prediction of a core's health state based on monitored information and taking action before the core gets damaged.

The intra-node monitoring of cores, D-TSU, and D-FDU is separated in two categories: time and event-driven. Time-driven messages are heartbeat messages that contain a set of core health information. Of particular interest are faults that influence the actual core performance. The D-FDU

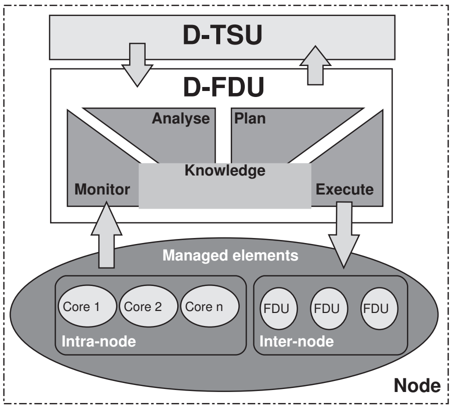

Figure 2: MAPE cycle of the D-FDU

expects a heartbeat message of a core in a certain time interval. If no heartbeat messages arrive at the D-FDU within the expected interval, the associated core will be suspected as faulty. When a permanent fault can be assured, e.g. multiple faults are detected in a short period of time; the D-FDU considers the core as completely broken. As a consequence the D-FDU considers the core as faulty and informs the D-TSU. The D-TSU itself is monitored by the D-FDU with the same techniques as a regular core. Thus, D-TSU faults can be detected as well.

The D-FDU communicates with the D-TSU via command messages, i.e. notify, request, and response messages. The D-TSU requests the D-FDU to change the frequency of a core or to reduce the frequency in the case of low workload, while the D-FDU reports the D-TSU on thermal and error conditions. In case of an intermittent or permanent error, the D-TSU temporarily or permanently stops scheduling any threads to the broken core.

D-FDUs can suffer from faults as well. To distribute reliability information between nodes, D-FDUs monitor each other.

Event-driven messages are alert messages in case of core faults. These messages are triggered by the L-FDU and notify the affiliated D-FDU within the node.

4.4 Machine Check Architecture

Most recent microprocessors are equipped with an architectural subsystem called Machine Check Architecture (MCA) [13] that is able to detect and correct certain faults. For instance for AMD K10 processor family [2], the MCA can detect faults in the data and instruction cache, the bus unit, the load-store unit, the northbridge, and the reorder buffers. In the case of a fault, MCA implementations mostly distinguish between two different types of faults:

1. Correctable faults that can be repaired on-the-fly. Therefore, the thread does not suffer from data and execution state corruption. These faults are handled by the core.

2. Non-correctable faults (mainly permanent faults and multiple bit flips) let the thread remain in a corrupted state.

We assume that all cores include a minimal Machine Check Architecture, which checks the fetched instructions and data for ECC checksum errors in registers, Frame Memory, and caches.

Since frequent occurrences of correctable and non-correctable faults may be a direct indicator for intermittent or permanent faults, or a permanent breakdown of the whole core, the L-FDUs transmit this information within its periodic heartbeat messages to the D-FDU. The D-FDU uses then the information to make predictions about the current reliability state of the core.

4.5 Control flow checking

Assuming control flow instructions within a dataflow thread, we incorporate a hybrid mechanism to dynamically detect control flow errors during runtime. Our proposed approach works as follows:

1. We instrument the application code of the dataflow threads at compile time by adding checkpoints (software instrumentation) to each basic block. A basic block consists of instructions and no control flow, i.e. branches, calls, or jumps. Additionally, it contains the maximum execution time until the next checkpoint is reached.

2. During runtime, a hardware check unit connected to the pipeline of a core reads the instrumented data in order to verify the correctness of the control flow. The checker unit compares the information from the actual control flow with the expected values from the instrumentation.

Double-executed threads (see Section 4.6) detect faulty behavior after their execution. The checker technique can speed up fault detection and permits lower detection latencies for transient and permanent errors affecting the control flow.

The proposed hybrid control flow checking approach combines benefits of both hard- and software mechanisms. The hardware overhead of this technique is limited to a small check unit with low complexity, while the overhead in execution time is caused by only few additional code instructions depending on the amount of executed control flow instructions within a thread.

4.6 Double execution

Our functional dataflow execution model simplifies the duplication of threads dynamically during the execution. We follow the definitions given by Rotenberg [26] and call the thread that is duplicated *leading thread* and its copy *trailing thread*. Please note that we use this terminology only to distinguish between threads, threads must not executed one after each.

Since the execution of dataflow threads is side-effect free and writes are only assigned once, we must only duplicate the *continuation* of a thread. This relaxes the complexity for the memory management as well as the management of the trailing thread.

Within a Thread-to-Core List (TCL) in each D-TSU, all continuations scheduled to a core within the node are redundantly stored. Our approach only duplicates the redundantly stored continuation in the D-FDU and schedules it to another core within the node. This means we can exploit data locality by sharing the thread frame between the leading and the trailing thread.

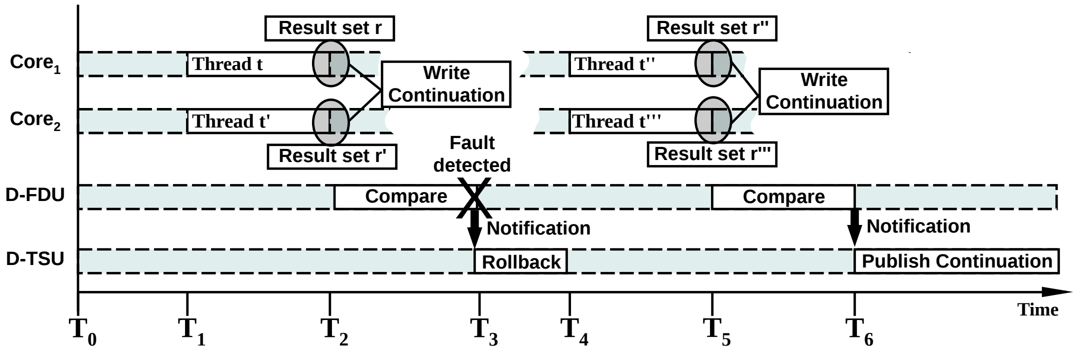

Figure 3: Example for a recovery scenario. Thread t is duplicated as t and t'. After the execution both thread instances write back their result set, which are compared by the D-FDU (at T_2). In the case of a transient fault, the D-TSU re-executes Thread t as t'' and t''' again on the same cores (T_4).

The L-FDUs reduce the result set per thread to a 32-bit signature and forward it to the node's D-FDU, which compares all signature pairs. The D-FDU signals the D-TSU the commitment of the leading thread. In the fault free case, the results of the leading thread are forwarded by the L-TSU to the D-TSU and stored in all consuming thread frames. Otherwise, the D-TSU has to trigger the recovery mechanism, described in Section 5. In more detail, double execution works as follows:

1. A thread is duplicated when its synchronization count becomes zero, i.e. a thread has received all its input values and is ready to execute. The L-TSU proceeds with the execution of the leading thread as usual.

2. To indicate the thread's duplication, the L-TSU sends notification messages to the D-TSU and the D-FDU. The D-TSU is responsible for copying the redundantly stored continuation of the thread and distributing it to the same or another core within the node, depending on which type of fault to detect. To detect transient faults, the D-TSU can schedule the thread to the same core. To detect permanent and intermittent faults as well, the D-TSU schedules the thread on a core within the same node, but on a different core by passing the copied continuation to the L-TSU of the core.

3. When both threads have finished execution, the L-TSU redirects the writes of the threads to the D-TSU and the D-FDU. The D-TSU manages a mechanism to buffer the writes until the D-FDU, which is in charge of comparing the results, gives a feedback.

4. In the case of a fault free execution the D-TSU deletes the continuation in its TCL and forwards the writes of the leading thread to the appropriate consuming threads. In the case of a fault, it has to re-execute the thread.

5. FAULT RECOVERY

The beauty of the dataflow execution model is side-effect free thread execution and single-assignment data passing between threads. This inherent functional semantic includes execution checkpoints between the dataflow threads. In other words, a dataflow thread can be restarted, as long as no writes to consumer threads have taken place. This is always the case in DTA-C, since the output frame becomes visible only after finishing the whole execution of the producer thread. Compared to a state-of-the-art many-core systems, these dataflow checkpoints promise a smaller memory footprint and simpler semantic for rollback-recovery mechanisms.

Figure 3 shows how the recovery mechanism will work. Note that we implicitly assume double execution to detect faults. When the D-FDU determines a fault within a monitored core (between time T_2 and T_3), it provides the corresponding core ID, together with the fault information to its affiliated D-TSU. Subsequently, the D-FDU tries to determine the cause of the detected fault. Depending on the kind of the fault the D-TSU can either restart the thread (at T_4, after the rollback between time T_3 and T_4) on the same core or re-allocate all threads of the faulty core to reliable cores. In the given case of a transient fault, usually the D-TSU will try to re-execute a thread again on the original core (at T_4). The re-execution can easily be done by overwriting the continuation field at the L-TSU with the redundant continuation field hold by the D-TSU. The L-TSU will then schedule the thread again.

In our approach restarting threads is assured by the D-TSU, which only forwards writes to the consuming thread frames if and only if the D-FDU signals the fault free execution of the producing thread.

If the D-FDU assumes a permanent or intermittent fault due to many re-execution attempts or information from the L-FDU, it must exclude the faulty core from further workload. This is done by providing the D-TSU with the information, which core is faulty. Consequently, the D-TSU re-schedules all threads of the faulty core on another reliable core. In order to do that, the D-TSU traverses its TCL and searches for corresponding entries regarding the faulty core. If the D-TSU finds an entry that is associated with the faulty core, it re-assigns the entry to a reliable core. Subsequently, the L-TSU has to allocate a thread frame for the newly assigned thread and fill the frame with the data from the D-TSU.

6. CONCLUSION

This paper presented a concept to cope with transient, intermittent, and permanent faults on all levels of a parallel hierarchical threaded dataflow system.

The detection of faults is done by control flow checking, double execution, and by exploiting the machine check architecture of the underlying cores. The hybrid control flow checking mechanism promises a low detection latency of program flow errors and low overhead concerning additional execution time. Permanent and intermittent faults can be recognized by scheduling the same threads onto different cores. Transient faults can be detected by executing the same threads on the same or different cores. Furthermore, we proposed a lean recovery mechanism, which exploits the thread-level checkpoints, intrinsic in our baseline dataflow architecture.

Acknowledgement

This work was partly funded by the European FP7 projects TERAFLUX (id. 249013), HiPEAC (IST-217068), and IT PRIN 2008 (200855LRP2). The authors wish to thank Dr. N. Puzovic and Z. Popovic for their initial studies of the DTA-C architecture.

7. REFERENCES

[1] International Technology Roadmap for Semiconductors 2007 Edition. Website. Available online at http://www.itrs.net, visited on August 12th 2011.

[2] AMD64 Architecture Programmer's Manual Volume 2: System Programming, 2006.

[3] Z. Alkhalifa, V. Nair, N. Krishnamurthy, and J. Abraham. Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):627–641, 1999.

[4] P. Bernardi, L. Bolzani, M. Rebaudengo, M. S. Reorda, F.L.Vargas, and M. Violante. A New Hybrid Fault Detection Technique for Systems-on-a-Chip. *IEEE Transactions on Computers*, 55(2):185–198, 2006.

[5] S. Borkar. Designing Reliable Systems from Unreliable Components: the Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, 2005.

[6] K. Echtle. Fault tolerance based on time-staggered redundancy. In *Fehlertolerierende Rechensysteme / Fault-Tolerant Computing Systems, 3. Internationale GI/ITG/GMA-Fachtagung*, pages 348–361, London, UK, 1987.

[7] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34:375–408, September 2002.

[8] T. Gautier, X. Besseron, and L. Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the International Workshop on Parallel Symbolic Computation (PASCO)*, pages 15–23, New York, NY, USA, 2007. ACM.

[9] R. Giorgi, Z. Popovic, and N. Puzovic. DTA-C: A Decoupled Multi-Threaded Architecture for CMP Systems. In *Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2007)*, pages 263–270, Los Alamitos, CA, USA, Oct. 2007. IEEE Computer Society.

[10] R. Giorgi, Z. Popovic, and N. Puzovic. Implementing Fine/Medium Grained TLP Support in a Many-Core Architecture. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5657 of *Lecture Notes in Computer Science*, pages 78–87. Springer Berlin / Heidelberg, 2009.

[11] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. Soft-Error Detection Using Control Flow Assertions. In *Proceedings of the 18th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, pages 581–588, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[12] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryk, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu. A design study of the EARTH multiprocessor. In *Proceedings of the IFIP WG10.3 working conference on Parallel Architectures and compilation techniques*, PACT '95, pages 59–68, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[13] R. Iyer, N. Nakka, Z. Kalbarczyk, and S. Mitra. Recent advances and new avenues in hardware-level reliability support. *IEEE Micro*, 25(6):18–29, 2005.

[14] S. Jafar, T. Gautier, A. Krings, and J. louis Roch. J.l.: A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In *Euro-Par*, pages 675–684, 2005.

[15] J. Kephart and D. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, Jan. 2003.

[16] A. Mahmood and E. McCluskey. Concurrent Error Detection Using Watchdog Processors–A Survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.

[17] T. Michel, R. Leveugle, and G. Saucier. A New Approach to Control Flow Checking without Program Modification. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS)*, pages 334–341, Los Alamitos, CA, USA, 1991. IEEE Computer Society.

[18] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 99–110, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[19] A. Nguyen-tuong, A. S. Grimshaw, and M. Hyett. Exploiting data-flow for fault-tolerance in a wide-area parallel system. In *Proceedings of the 15th International Symposium on Reliable and Distributed Systems*, pages 1–11. IEEE Computer Society, 1996.

[20] N. Oh, P. Shirvani, and E. McCluskey. Control-flow Checking by Software Signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002.

[21] J. Ohlsson and M. Rimen. Implicit Signature Checking. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 218–227, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

[22] A. Portero, Z. Yu, and R. Giorgi. T-Star (T*): An x86-64 ISA Extension to support thread execution on many cores. pages pp. 277–280. HiPEAC ACACES-2011, July 2011.

[23] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 111–122. IEEE Computer Society, 2002.

[24] R. Ragel and S. Parameswaran. A Hybrid Hardware–Software Technique to Improve Reliability in Embedded Processors. *ACM Transactions on Embedded Computing Systems*, 10(3):36:1–36:16, 2011.

[25] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[26] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. *International Symposium on Fault-Tolerant Computing*, 0:84–91, 1999.

[27] M. Schuette and J. Shen. Processor Control Flow Monitoring Using Signatured Instruction Streams. *IEEE Transactions on Computers*, 36(3):264–276, 1987.

[28] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems (2nd ed.): design and evaluation*. Digital Press, Newton, MA, USA, 1992.

[29] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 123–134, Washington, DC, USA, 2002. IEEE Computer Society.

[30] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 177–186, Washington, DC, USA, 2004. IEEE Computer Society.

[31] K. Stavrou, P. Evripidou, and P. Trancoso. DDM-CMP: Data-Driven Multithreading on a Chip Multiprocessor. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 3553, pages 364–373. Springer, Berlin, Heidelberg, 2005.

[32] S. Weis, A. Garbade, S. Schlingmann, and T. Ungerer. Towards Fault Detection Units as an Autonomous Fault Detection Approach for Future Many-Cores. In *ARCS 2011 Workshop Proceedings*, pages 20–23. VDE Verlag, Feb. 2011.

[33] K. Wilken and J. Shen. Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(6):629–641, 1990.

[34] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "codelet" program execution model for exascale machines: position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.