

# Adaptive Fault Tolerance for Many-Core Based Space-Borne Computing

Mark James  
Jet Propulsion Laboratory,  
California Institute of  
Technology  
Pasadena, CA  
mjames@jpl.nasa.gov

Paul Springer  
Jet Propulsion Laboratory,  
California Institute of  
Technology  
Pasadena, CA  
pls@jpl.nasa.gov

Hans Zima  
Jet Propulsion Laboratory,  
California Institute of  
Technology, Pasadena, CA  
zima@jpl.nasa.gov

## Keywords

many-core systems, dependability, fault tolerance, space missions, autonomy

## ABSTRACT

*This paper describes an approach to providing software fault tolerance for future deep-space robotic NASA missions, which will require a high degree of autonomy supported by an enhanced on-board computational capability. Such systems have become possible as a result of the emerging many-core technology, which is expected to offer 1024-core chips by 2015. We discuss the challenges and opportunities of this new technology, focusing on introspection-based adaptive fault tolerance that takes into account the specific requirements of applications, guided by a fault model. Introspection supports runtime monitoring of the program execution with the goal of identifying, locating, and analyzing errors. Fault tolerance assertions for the introspection system can be provided by the user, domain-specific knowledge, or via the results of static or dynamic program analysis. This work is part of an on-going project at the Jet Propulsion Laboratory in Pasadena, California.*

## 1. INTRODUCTION

On-board computing systems for space missions are subject to stringent dependability requirements, with enforcement strategies focusing on strict and widely formalized design, development, verification, validation, and testing procedures. Nevertheless, history has shown that despite these precautions errors occur, sometimes resulting in the catastrophic loss of an entire mission. There are theoretical as well as practical reasons for this situation:

1. No matter how much effort is spent for verification and test well-known undecidability and NP-completeness results show that many relevant problems are either undecidable or computationally intractable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

2. As a result, large systems typically *do* contain design faults.
3. Even a perfectly designed system may be subject to external faults, such as radiation effects and operator errors.

As a consequence, it is essential to provide methods that avoid system failure and maintain the functionality of a system, possibly with degraded performance, even in the case of faults. This is called *fault tolerance*.

Fault tolerant systems were built long before the advent of the digital computer, based on the use of replication, diversified design, and federation of equipment. In an article on Babbage's difference engine published in 1834 Dionysius Lardner wrote [14]: "The most certain and effectual check upon errors which arise in the process of computation is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computation by different methods." An example for an early fault-tolerant computer is NASA's *Self-Testing-and-Repairing (STAR)* system developed for a 10-year mission to the outer planets in the 1960s. Today, highly sophisticated fault-tolerant computing systems control the new generation of fly-by-wire aircraft, such as the Airbus and Boeing airliners. Perhaps the most widespread use of fault-tolerant computing has been in the area of commercial transactions systems, such as automatic teller machines and airline reservation systems.

Most space missions of the past were largely controlled from Earth, so that a significant number of failures could be handled by putting the spacecraft in a "safe" mode, with Earth-bound controllers attempting to return it to operational mode. This approach will no longer work for future deep-space missions, which will require enhanced autonomy and a powerful on-board computational capability. Such missions are becoming possible as a result of recent advances in microprocessor technology, which are leading to low-power many-core chips that today already have on the order of 100 cores, with 2015 technology expected to offer 1024-core systems. These developments have many consequences for fault tolerance, some of them challenging and others providing new opportunities. In this paper we focus on an approach for software-implemented application-adaptive fault tolerance. The paper is structured as follows: In Section 2, we establish a conceptual basis, providing more precise definitions for the notions of dependability and fault tolerance. Section 3 gives an overview of future missions

and their requirement, and outlines an on-board architecture that complements a radiation-hardened spacecraft control and communication component with a COTS-based high-performance processing system. After introducing introspection in Section 4, we discuss introspection-based adaptive fault tolerance in Section 5. The paper ends with an overview of related work and concluding remarks in Sections 6 and 7.

## 2. FAULT TOLERANCE IN THE CONTEXT OF DEPENDABILITY

**Dependability** has been defined by the IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance as the “*trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers*”. Dependability is characterized by its *attributes*, the *threats* to it, and the *means* by which it can be achieved [2, 3].

The *attributes* of dependability specify a set of properties that can be used to assess how a system satisfies its overall requirements. Key attributes are reliability, availability, mean-time-to-failure, and safety. A *threat* is any fact or event that negatively affects the dependability of a system. Threats can be classified as faults, errors, or failures. Their relationship can be illustrated by the *fault-error-failure* chain shown in Figure 1.

A **fault** is a defect in a system. Faults can be *dormant*—e.g., incorrect program code that is not executed—and have no effect. When *activated* during system operation, a fault leads to an **error**, which is an illegal system state. Errors may be propagated through a system, generating other errors. For example, a faulty assignment to a variable may result in an error characterized by an illegal value for that variable; the use of the variable for the control of a for-loop can lead to ill-defined iterations and other errors, such as illegal accesses to data sets and buffer overflows. A **failure** occurs if an error reaches the service interface of a system, resulting in system behavior that is inconsistent with its specification.

The execution of a system can be modeled by a sequence of states, with state transitions being caused as the result of atomic *actions*. In a first approximation, the set of all system states can be partitioned into **correct states** and **error states**. By separating those error states that allow a recovery from those that represent system failure the set of all error states is further partitioned into **tolerated error states** and **failure states** (Figure 2). The transitions between these state categories can be described by classifying unambiguously each action as a *correct*, *fault*, or *recovery* action [5]:

- A correct action, executed in a correct state, results in a correct state
- A fault action, executed in a correct state, results in an error state (tolerated or failure)
- A correct or fault action, executed in a tolerated error state, results in an error state (tolerated or failure)
- A recovery action, executed in a tolerated error state, results in a correct state.

With the above terminology in place, we can now precisely characterize a system as **fault tolerant** if it never enters a

failure state. Errors may occur in such a system, but they never reach its service boundary and always allow recovery to take place. The implementation of fault tolerance in general implies three steps: error detection, error analysis, and recovery.

The *means* for achieving dependability include fault prevention, fault removal, and fault tolerance. *Fault prevention* addresses methods that prevent faults to being incorporated into a system. In the software domain, such methods include restrictive coding structures that avoid common programming faults, the use of object-oriented techniques, and the provision of high-level APIs. An example for hardware fault prevention is shielding against radiation-caused faults. *Fault removal* refers to the a set of techniques that eliminate faults during the design and development process. *Verification and Validation (V&V)* are important in this context: Verification provides methods for the review, inspection, and test of systems, with the goal of establishing that they conform to their specification. Validation checks the specification in order to determine if it correctly expresses the needs of the system’s users.

For theoretical as well as practical reasons, neither fault prevention nor fault removal provide complete solutions, i.e., in general for non-trivial programs there is no guarantee that they do not contain design faults. However, even in a program completely free of design faults, hardware malfunction can cause software errors at execution time. In the domain underlying this paper, the problem is actually more severe: a spacecraft can be hit by radiation, which can cause arbitrary errors in data and control structures. This is discussed in more detail in the next section.

## 3. FUTURE SPACE MISSIONS AND THEIR REQUIREMENTS

Future deep-space missions face the challenge of designing, building, and operating progressively more capable autonomous spacecraft and planetary rovers. Given the latency and bandwidth of spacecraft-Earth communication for such missions, the need for enhanced autonomy becomes obvious: Earth-based mission controllers will be unable to directly control distant spacecraft and robots to ensure timely precision and safety, and to support “opportunistic science” by capturing rapidly changing events, such as dust devils on Mars or volcanic eruptions on a remote moon in the solar system [4]. Furthermore, the high data volume yielded by smart instruments on board of the spacecraft can overwhelm the limited bandwidth of spacecraft-Earth communication, enforcing on-board data analysis, filtering, and compression. Science processing will require a high-performance capability that may range up to hundreds of Teraops for on-board synthetic aperture radar (SAR), hyperspectral assessment of scenes, or stereo vision. Currently, the performance of traditional mission architectures lags that of commercial products by at least two orders of magnitude; furthermore, this gap is expected to widen in the future. As a consequence, the traditional approach to on-board computing is not expected to scale with the requirements of future missions. A radical departure is necessary.

Emerging technology offers a way out of this dilemma. Recent developments in the area of commercial multi-core architectures have resulted in simpler processor cores, en-

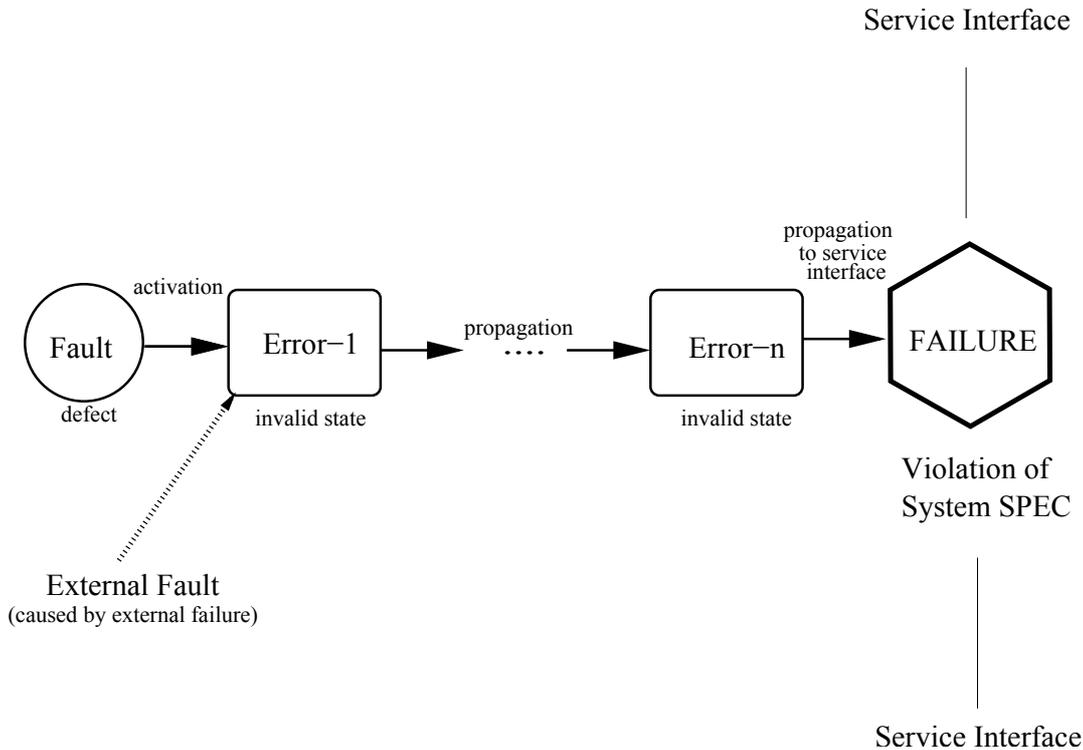


Figure 1: Threats: the fault-error-failure chain

hanced efficiency in terms of performance per Watt, and a dramatic increase in the number of cores on a chip, as illustrated by Tiler Corporation’s *Tile64* [22]—a homogeneous parallel chip architecture with 64 identical cores arranged in an 8x8 grid performing at 192 Gops with a power consumption of 170-300mW per core—or Intel’s terachip announced for 2011—an 80-core chip providing 1.01 Teraflops based on a frequency of 3.16 GHz, with a power consumption of 62W.

These trends suggest a new paradigm for spacecraft architectures, in which the ultra-reliable radiation-hardened core component responsible for control, navigation, data handling, and communication is extended with a scalable commodity-based multi-core system for autonomy and science processing. This approach will provide the basis for a powerful parallel on-board supercomputing capability. However, bringing COTS components into space leads to a new problem—the need to address their vulnerability to hardware as well as software faults.

Space missions are subject to faults caused by equipment failure or environmental impacts, such as radiation, temperature extremes, or vibration. Missions operating close to the Earth/Moon system can be controlled from the ground. Such missions may allow **controlled failure**, in the sense that they fail only in specific, pre-defined modes, and only to a manageable extent, avoiding complete disruption. Rather than providing the capability of resuming normal operation, a failure in such a system puts it into a *safe mode*, from which recovery is possible after the failure has been detected and identified.

As an example, the on-board software controlling robotic planetary exploration spacecraft for those portions of a mis-

sion during which there is no critical activity (such as de-tumbling the spacecraft after launch or descent to a planetary surface) can be organized as a system allowing controlled failure. When a fault is detected during operation, all active command sequences are terminated, components inessential for spacecraft survival are powered off, and the spacecraft is positioned into a stable sun-pointed attitude. Critical information regarding the state of the spacecraft and the fault are transmitted to ground controllers via an emergency link. Restoring the spacecraft health is then delegated to controllers on Earth.

However, a fail-safe approach is *not* adequate for deep-space missions beyond immediate and continuous control from the Earth. For such missions, fault tolerance is a key prerequisite, i.e., a *fail-operational* response to faults must be provided, implying that the spacecraft must be able to deal autonomously with faults and continue to provide the full range of critical functionality, possibly at the cost of degraded performance. Systems which preserve continuity of service can be significantly more difficult to design and implement than fail-controlled systems. Not only is it necessary to determine that a fault has occurred, the software must be able to determine the effects of the fault on the system’s state, remove the effects of the fault, and then place the system into a state from which processing can proceed.

This is the situation on which the rest of this paper is based. We focus on strategies and techniques for providing *adaptive, introspection-based fault tolerance* for space-borne systems.

Deep-space missions are subject to radiation in the form of cosmic rays and the solar wind, exposing them to pro-

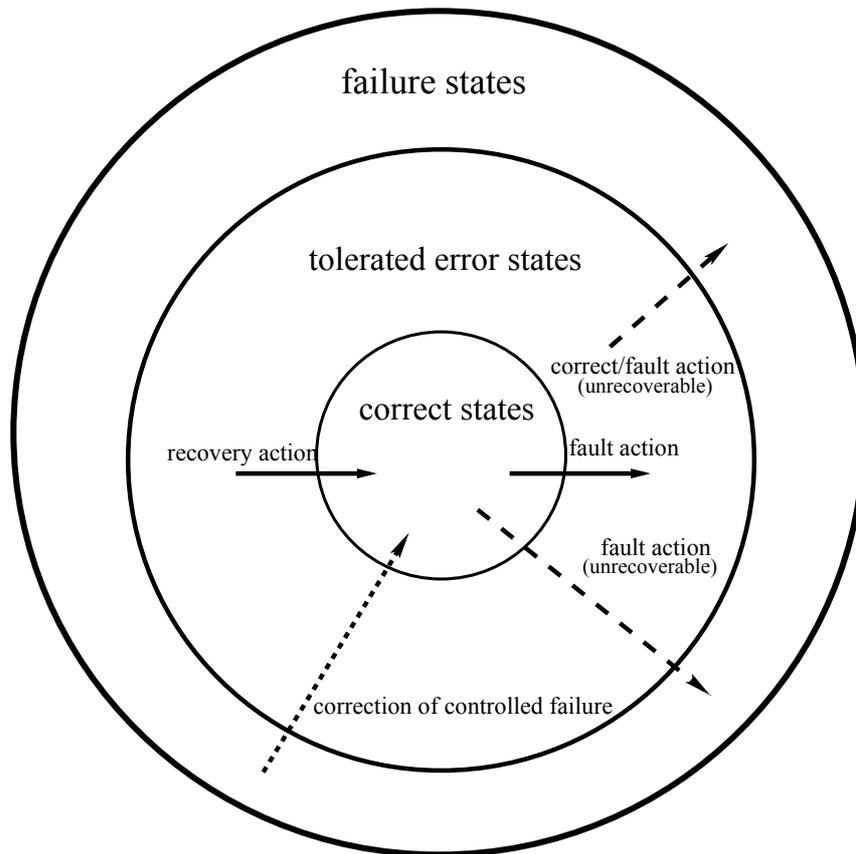


Figure 2: State space partitioning

tons, alpha particles, heavy ions, ultraviolet radiation, and X-rays. Radiation can interact with matter through atomic displacement—a rearrangement of atoms in a crystal lattice—or ionization, with the potential of causing permanent or transient damage [19]. Modern COTS circuits are protected against long-term cumulative degradation as well as catastrophic effects caused by radiation. However, they are exposed to *transient faults* in the form of Single Event Upsets or Multiple Bit Upsets, which do not cause lasting damage to the device. A *Single Event Upset (SEU)* changes the state of a single bit in a register or memory, whereas a *Multiple Bit Upset (MBU)* results in a change of state of multiple adjacent bits. The probability of SEUs and MBUs depends on the environment in which the spacecraft is operating, and on the detailed characterization of the hardware components in use. COTS semiconductor fabrication processes vary: with the 65nm process now in commercial production, some of the semiconductor foundries are using Silicon on Insulator (SOI) construction, which makes the chips less susceptible to these radiation effects.

Depending on the efficacy of fault tolerance mechanisms, SEUs and MBUs can manifest themselves at different levels. For example, faults may affect processor cores and caches, DRAM memory units, memory controllers, on-chip communication networks, I/O processors nodes, and inter-connection networks. This can result in the corruption of instruction fetch/decode, address selection, memory units, synchronization, communication, and signal/interrupt pro-

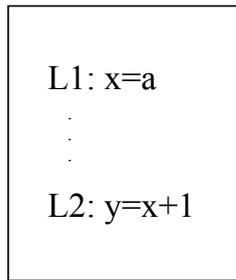
cessing. In a sequential thread this may lead to the (unrecognized) use of corrupted data and the execution of wrong or illegal instructions, branches, and data accesses in the program. Hangs or crashes of the program, as well as unwarranted exceptions are other possible consequences. In a distributed system, transient faults can cause communication errors, livelock, deadlock, data races, or arbitrary Byzantine failures [13].

Some of these effects may be caught and corrected in the hardware (e.g., via the use of an error-correcting code (ECC)) with no disruption of the program. A combination of hardware and software mechanisms may provide an effective approach, as in the case of the fault isolation of cores in a multi-core chip [1]. Other faults, such as those causing illegal instruction codes, illegal addresses, or the violation of access protections may trigger a synchronous interrupt, which can lead to an application-specific response. In a distributed system, watchdogs may detect a message failure.

However, in general, an error may remain undetected. Consider the simple example of the possible effect a transient fault may have (Figure 3): it illustrates an illegal read access to a variable caused by the corruption of an assignment. Assume that the assignment in *L1* is the only definition for *x* reaching *L2*, and that an SEU destroys that assignment, for example by changing its target from *x* to another variable. Then the use of *x* in statement *L2* is undefined.

Figure 4 outlines key building blocks of an architecture for

## Original Program



SEU  
→

## Corrupted Program

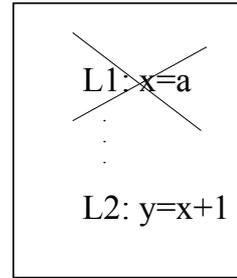


Figure 3: Illegal variable access as a result of an SEU

space-borne computing in which the radiation-hardened core is augmented with a COTS-based scalable high-performance computing system (HPCS).

The *Spacecraft Control & Communication System* is the core component of the on-board system, controlling the overall operation, navigation, and communication of the spacecraft. Due to its critical role for the operation and survival of the spacecraft this system is typically implemented using radiation-hardened components that are largely immune to the harsh radiation environments encountered in space.

The *Fault-Tolerant High-Capability Computational Subsystem (FTCS)* is designed to provide an additional layer of fault tolerance around the HPCS via a *Reliable Controller* that shields the Spacecraft Control & Communication System from faults that evaded detection or masking in the High Performance Computing System. The *Reliable Controller* is the only component of the FTCS that communicates directly with the spacecraft control and communication system. As a consequence, it must satisfy stringent reliability requirements. Important approaches for implementing the Reliable Controller—either on a pure software basis, or by a combination of hardware and software—have been developed in the Ghidrah [15] and ST8 systems [18].

## 4. INTROSPECTION

The rest of this paper deals with an introspection-based approach to providing fault tolerance for the High Performance Computing System in the architecture depicted in Figure 4. A generic framework for introspection has been described in [9]; here we outline its major components.

Introspection provides a generic software infrastructure for the monitoring, analysis, and feedback-oriented management of applications at *execution time*.

Consider a parallel application executing in the High Performance Computing System. Code and data belonging to the object representation of the application will be distributed across its components, creating a partitioning of the application into *application segments*. An instance of the introspection system consists of a set of interacting **introspection modules**, each of which can be linked to application segments. The structure of an individual introspection module is outlined in Figure 5. Its components include:

- **Application/System Links** are either *sensors* or *actuators*. *Sensors* represent hardware or software events

that occur during the execution of the application; they provide input from the application to the introspection module. *Actuators* represent feedback from the introspection module to the application. They are triggered as a result of module-internal processing and may result in changes of the application state, its components, or its instrumentation.

- **Inference Engine** The nature of the problems to which introspection is applied demands efficient and flexible control of the associated application segments. These requirements are met in our system by the *Spacecraft Health Inference Engine (SHINE)* as the core of each introspection module. SHINE is a real-time inference engine that provides an expert systems capability and functionality for building, accessing, and updating a structured knowledge base.
- The **Knowledge Base** consists of declarative facts and rules that specify how knowledge can be processed. It may contain knowledge about the underlying system, the programming languages supported, the application domain, and properties of application programs and their execution that are either derived by static or dynamic analysis or supplied by the user.
- **Agents** The functionality of a module can be implemented by a system of asynchronously operating autonomous agents with internal state. The distributed nature of the overall system and its implementation using autonomous agents enable local analysis without the need to always refer to a centralized entity providing full global knowledge.

We have implemented a prototype introspection system for a cluster of Cell Broadband Engines. Figure 6 illustrates the associated hierarchy of introspection modules, where the levels of the hierarchy, from bottom to top, are respectively associated with the SPEs, the PPE, and the overall cluster.

## 5. ADAPTIVE FAULT TOLERANCE FOR HIGH-PERFORMANCE ON-BOARD COMPUTING

The prototype system mentioned above relied on user-specified assertions guiding the introspection system. In the

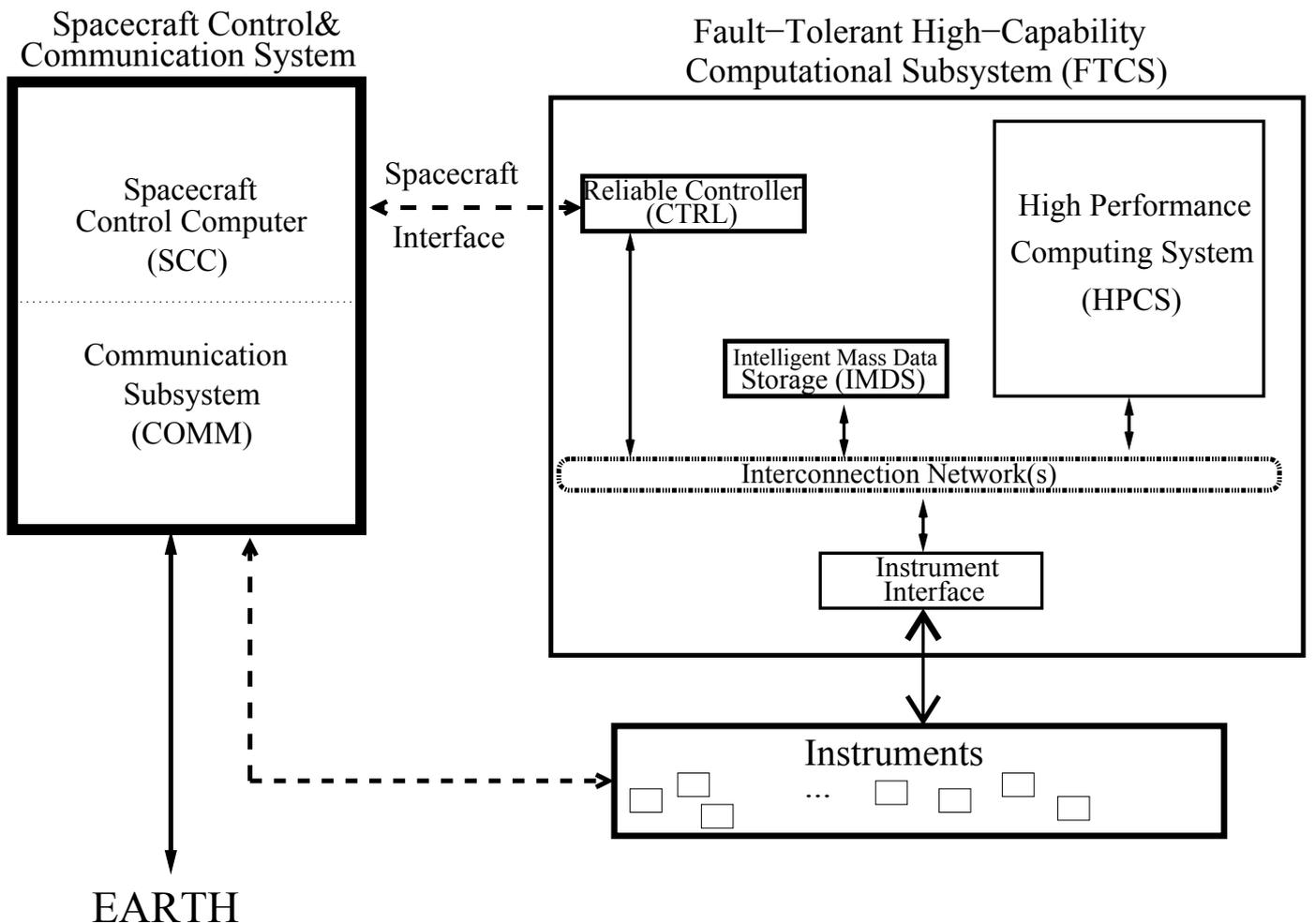


Figure 4: An architecture for scalable space-borne computing

following we outline the ideas underlying our current work, which generalizes this system in a number of ways, with a focus on providing support for the automatic generation of assertions.

Our approach to providing fault tolerance for applications executing in the high-performance computing system is adaptive in the sense that faults can be handled in a way that depends on the potential damage caused by them. This enables a flexible policy resulting in a reduced performance penalty for the fault tolerance strategy when compared to fixed-redundancy schemes. For example, an SEU causing a single bitflip in the initial phase of an image processing algorithm may not at all affect the outcome of the computation. However, SEU-triggered faults such as the corruption of a key data structure caused by an illegal assignment to one of its components, the change of an instruction code, or the corruption of an address computation may have detrimental effects on the outcome of the computation. Such faults need to be handled through the use of redundancy, with an approach that reflects their severity and takes into account known properties of the application and the underlying system.

## 5.1 Assertions

An *assertion* describes a propositional logic predicate that must be satisfied at certain locations of the program, during specific phases of execution, or in program regions such as loops and methods. Its specification consists of four components—the specification of an assertion expression, the region in which this expression can be applied, the characterization of the fault if the assertion is violated, and an optional recovery specification. We illustrate this by a set of examples.

**assert**  $((A(i) \leq B(i))$  **in**  $(L1)$  **fault**  $(F1, i, \dots)$  **recovery**  $(\dots)$   
 The assertion expression  $A(i) \leq B(i)$  must be satisfied immediately after the execution of the statement labeled by  $L1$ . If it fails, a fault type,  $F1$ , is specified and a set of relevant arguments is relayed to the introspection system. Furthermore, a hint for the support of a recovery method is provided.  $\square$

**assert**  $(x \neq 0)$  **pre in**  $(L2)$  **fault**  $(FT2, x, \dots)$   
**assert**  $(z = f2(x))$  **in**  $(L2)$  **fault**  $(FT3, x, y, z, \dots)$   
 The two assertion expressions  $x \neq 0$  and  $z = f2(x)$  respectively serve as precondition and postcondition for a state-

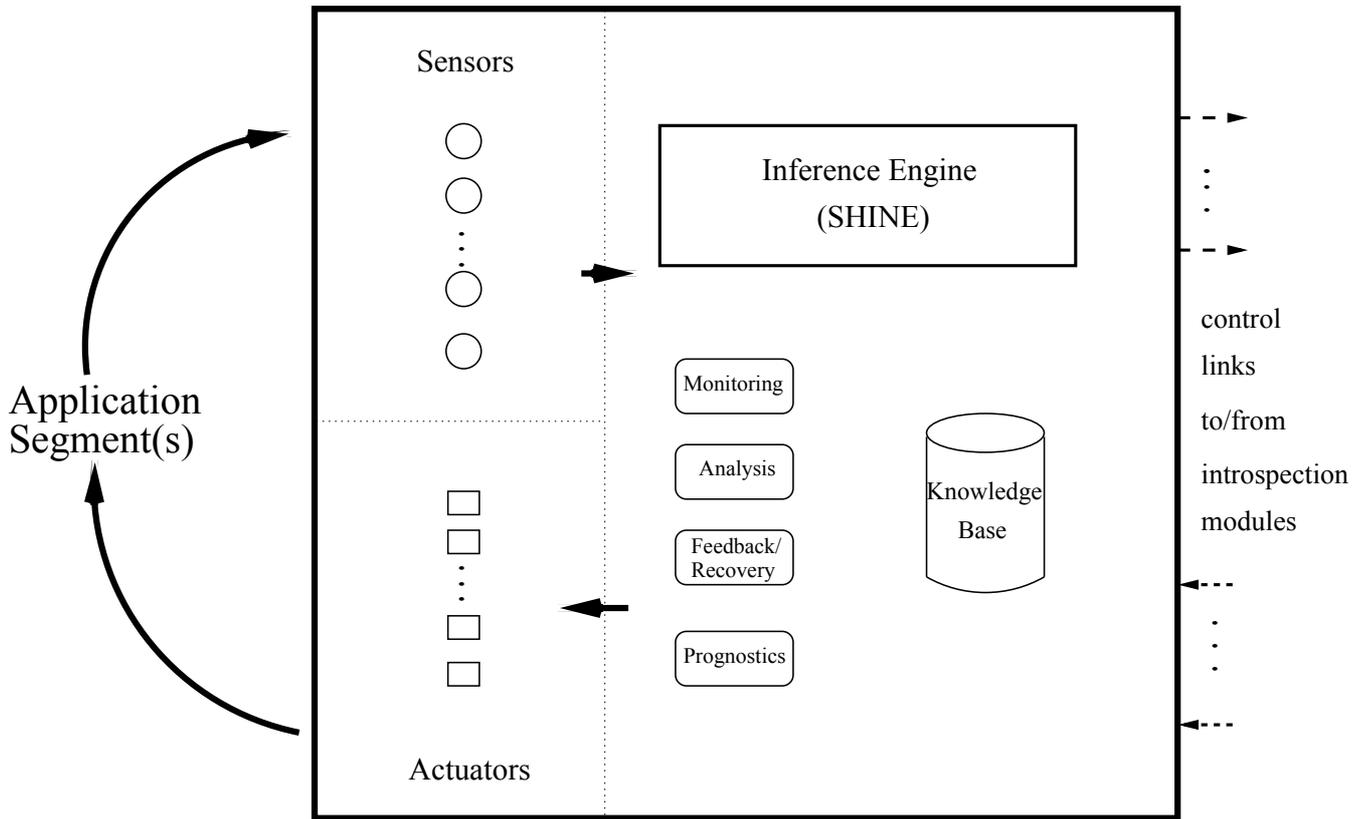


Figure 5: Introspection module

ment at label  $L2$ , with respective fault types  $FT2$  and  $FT3$  for assertion violations.  $\square$

**assert** ( $diff \geq \epsilon$ ) **invariant in** ( $r\_loop$ ) **fault** (...)

The assertion expression  $diff \geq \epsilon$  specifies an invariant that is associated with the region defined by  $r\_loop$ . It must be satisfied at any point of execution within this loop.  $\square$

## 5.2 Fault Detection and Recovery

Introspection-based fault tolerance provides a flexible approach that in addition to applying innovative methods can leverage existing technology.

Methods that are useful in this context include *assertion-based acceptance tests* that check the value of an assertion and transfer control to the introspection system in case of violation, and *fault detectors* that can effectively mask a fault by using redundant code based on analysis information (see Section 5.3).

Furthermore, faults in critical sections of the code can be masked by leveraging fixed redundancy techniques such as TMR or NMR. Another technique is the replacement of a function with an equivalent version that implements Algorithm-Based Fault Tolerance (ABFT).

Information supporting the generation of assertion-based acceptance tests as well as fault detectors can be derived from static or dynamic automatic program analysis, retrieved from domain- or system specific information contained in the knowledge base or be directly specified by an expert user. Figure 7 provides an informal illustration of the different

methods used to gather such information.

## 5.3 Analysis-Based Assertion Generation

Automatic analysis of program properties relevant for fault tolerance can be leveraged from a rich spectrum of existing tools and methods. This includes the static analysis of the control and data structures of a program, its intra- and inter-procedural control flow, data flow and data dependences, data access patterns, and patterns of synchronization and communication in multi-threaded programs [25, 17]. Other static tools can check for the absence of deadlocks or race conditions. Profiling from simulation runs or test executions can contribute information on variable ranges, loop counts, or potential bottlenecks [7]. Furthermore, dynamic analysis provides knowledge that cannot be derived at compile time, such as the actual paths taken during a program execution and dynamic dependence relationships.

Consider a simple example. In data flow analysis, a *use-definition chain* is defined as the link between the statement that *uses* (i.e., reads) a variable to the set of all *definitions* (i.e., assignments) of that variable that can reach this statement along an execution path. Similarly, a definition-use chain links a definition to all its uses. An SEU can break such a chain, for example by redirecting an assignment. This can result in a number of different faults, including the following:

- attempt to use an undefined variable or dereference an

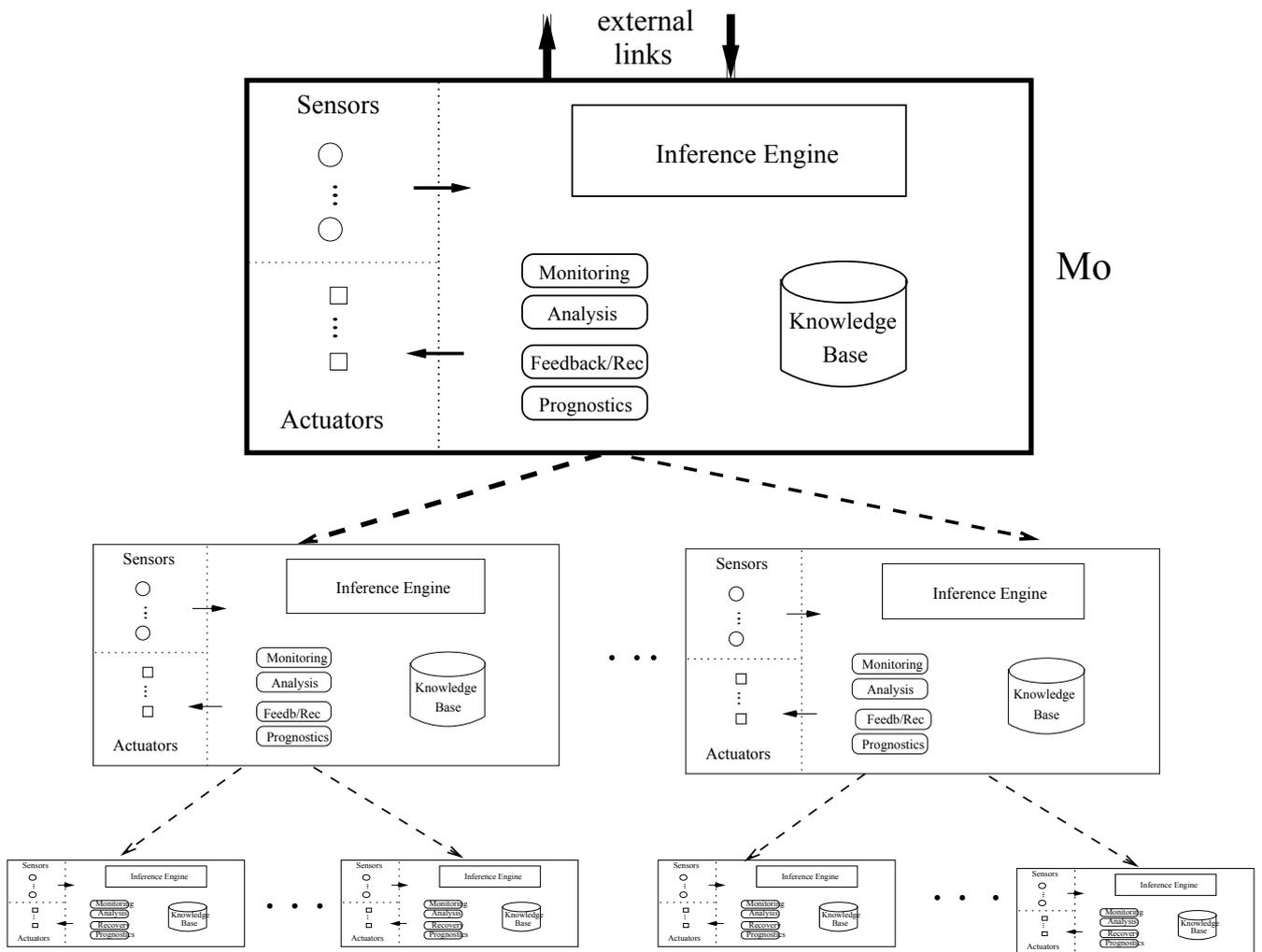


Figure 6: Introspection hierarchy for a cluster of Cell Broadband Engines

undefined pointer

- rendering a definition of a variable useless
- leading to an undefined expression evaluation
- destroying a loop bound

The results of static analysis (as well as results obtained from program profiling) can be exploited for fault detection and recovery in a number of ways, including the generation of assertions in connection with specific program locations or program regions. Examples include asserting:

- the value of a variable that has been determined to be a constant
- the value range of a variable or a pointer
- the preservation of use-definition and definition-use chains
- the preservation of dependence relationships
- a limit for the number of iterations in a loop

- an upper limit for the size of a data structure
- correctness of access sequences to files

The generation of such assertions must be based on the statically derived information in combination with the generation of code that records the corresponding relationships at runtime. A more elaborate technique that exploits static analysis for the generation of a fault detector using redundant code generation can be based on *program slicing* [23]. This is an analysis technique that extracts from a program the set of statements that affect the values required at a certain point of interest. For example, it answers the question which statements of the program contribute to the value of a critical variable at a given location. These statements form a *slice*. The occurrence of SEUs can disrupt the connection between a variable occurrence and its slice. A *fault detector* for a specific variable assignment generates redundant code based only on that slice, and compares its outcome with that of the original code.

Some of the techniques applied to sequential programs can be generalized to deal with multi-threaded programs. Of

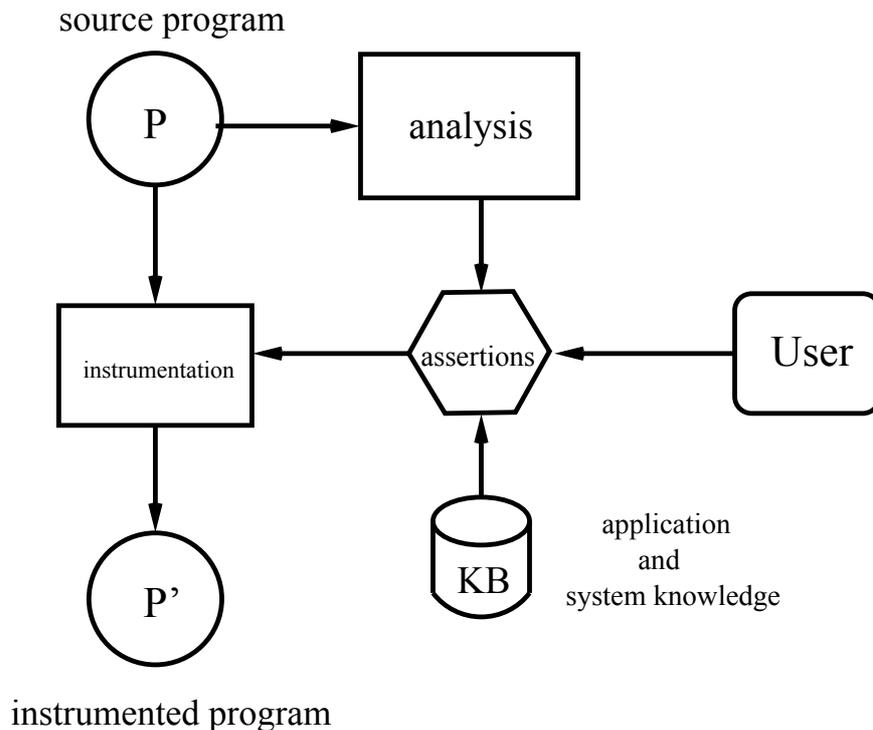


Figure 7: Assertion generation

specific importance in this context are programs whose execution is organized as a data-parallel set of threads according to the *Single-Program-Multiple-Data (SPMD)* paradigm since the vast majority of parallel scientific applications belong to this category [21].

## 6. RELATED WORK

The *Remote Exploration and Experimentation (REE)* [20] project conducted at NASA was among the first to consider putting a COTS-based parallel machine into space and address the resulting problems related to application-adaptive fault tolerance [11]. More recently, NASA's Millenium ST-8 project [18] developed a "*Dependable Multiprocessor*" around a COTS-based cluster using the IBM PowerPC 750FX as a data processor, with a Xilinx VirtexII 6000 FPGA co-processor for the support of application-specific modules for digital signal processing, data compression, and vector processing. A centralized system controller for the cluster is implemented using a redundant configuration of radiation-hardened Motorola processors.

Some significant work has been done in the area of assertions. The EAGLE system [6] provides an assertion language with temporal constraints. The *Design for Verification (D4V)* [16] system uses *dynamic assertions*, which are objects with state that are constructed at design time and tied to program objects and locations. Language support for assertions and invariants has been provided in Java 1.4, Eiffel for pre- and post condition in Hoare's logic, and the Java Modeling Language (JML). Intelligent resource management in an introspection-based approach has been proposed in [12].

Finally, the concept of introspection, as used in our work,

has been outlined in [24, 10]. A similar idea has been used by Iyer and co-workers for application-specific security [8] based on hardware modules embedded in a reliability and security engine.

## 7. CONCLUSION

This paper focused on software-provided fault tolerance for future deep-space missions providing an on-board COTS-based computing capability for the support of autonomy. We described the key features of an introspection framework for runtime monitoring, analysis, and feedback-oriented recovery, and outlined methods for the automatic generation of assertions that trigger key actions of the framework. A prototype version of the system was originally implemented on a cluster of Cell Broadband Engines; currently, an implementation effort is underway for the Tile64 system.

Future work will address an extension of the introspection technology to performance tuning and power management. Furthermore, we will study the integration of introspection with traditional V&V.

## Acknowledgment

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration and funded through the internal Research and Technology Development program.

## 8. REFERENCES

- [1] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith. Isolation in

- Commodity Multicore Processors. *IEEE Computer*, 40(6):49–59, June 2007.
- [2] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental Concepts of Dependability. Technical report, UCLA, 2000. CSD Report No. 010028.
- [3] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1), January-March 2004.
- [4] Rebecca Castano, Tara Estlin, Robert C. Anderson, Daniel M. Gaines, Andres Castano, Benjamin Bornstein, Caroline Chouinard, and Michele Judd. OASIS: Onboard Autonomous Science Investigation System for Opportunistic Rover Science. *Journal of Field Robotics*, 24(5):379–397, 2007.
- [5] Felix C. Gärtner. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM Computing Surveys*, 31(1):1–26, 1999.
- [6] Allen Goldberg, Klaus Havelund, and Conor McGann. Runtime Verification for Autonomous Spacecraft Software. In *Proceedings 2005 IEEE Aerospace Conference*, pages 507–516, March 2005.
- [7] Klaus Havelund and Allen Goldberg. Verify Your Runs. In *Proceedings Verified Software: Theories, Tools, Experiments (VSTTE'05)*, October 2005.
- [8] Ravishankar K. Iyer, Zbigniew Kalbarczyk, Karthik Pattabiraman, William Healey, Wen-Mei W. Hwu, Peter Klemperer, and Reza Farivar. Toward Application-Aware Security and Reliability. *IEEE Security and Privacy*, 5(1):57–62, 2007.
- [9] Mark James, Andrew Shapiro, Paul Springer, and Hans Zima. Adaptive Fault Tolerance for Scalable Cluster Computing in Space. *International Journal of High Performance Computing Applications (IJHPCA)*, 23(3), 2009. SAGE Publications.
- [10] Mark L. James and Hans P. Zima. An Introspection Framework for Fault Tolerance in Support of Autonomous Space Systems. In *Proceedings 2008 IEEE Aerospace Conference*, March 2008.
- [11] Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, Saurabh Bagchi, and Keith Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):560–579, 1999.
- [12] Dong-In Kang, Jinwoo Suh, Janice O. McMahon, and Stephen P. Crago. Preliminary Study toward Intelligent Run-time Resource Management Techniques for Large Multi-Core Architectures. In *Proceedings of the 2007 Workshop on High Performance Embedded Computing (HPEC07)*, September 2007.
- [13] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Programming Languages and Systems*, 4(3):382–401, July 1982.
- [14] Dionysius Lardner. Babbages’s Calculating Engine. *Edinburgh Review*, July 1834. Reprinted in P.Morrison and E.Morrison (Editors): Charles Babbage and His Calculating Engines, Dover, New York, 1961.
- [15] Ming Li, Wenchao Tao, Daniel Goldberg, Israel Hsu, and Yuval Tamir. Design and Validation of Portable Communication Infrastructure for Fault-Tolerant Cluster Middleware. In *Cluster’02: Proceedings of the IEEE International Conference on Cluster Computing*, page 266, Washington, DC, USA, September 2002. IEEE Computer Society.
- [16] Peter C. Mehrlitz and John Penix. Design for Verification with Dynamic Assertions. In *Proceedings of the 2005 29th Annual IEEE/NASA Software Engineering Workshop (SEW’05)*, 2005.
- [17] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, Berlin, Heidelberg, New York, 1999.
- [18] J. Samson, G. Gardner, D. Lupia, M. Patel, P. Davis, V. Aggarwal, A. George, Z. Kalbarczyk, and R. Some. High Performance Dependable Multiprocessor II. In *Proceedings 2007 IEEE Aerospace Conference*, pages 1–22, March 2007.
- [19] Philip P. Shirvani. Fault-Tolerant Computing for Radiation Environments. Technical Report 01-6, Center for Reliable Computing, Stanford University, Stanford, California 94305, June 2001. Ph.D. Thesis.
- [20] R. Some and D. Ngo. REE: A COTS-Based Fault Tolerant Parallel Processing Supercomputer for Spacecraft Onboard Scientific Data Analysis. In *Proceedings of the Digital Avionics Systems System Conference*, pages 7.B.3–1–7.B.3–12, 1999.
- [21] Michelle M. Strout, Barbara Kreaseck, and Paul Hovland. Data Flow Analysis for MPI Programs. In *Proceedings of the 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’06)*, June 2006.
- [22] Tile64 Processor Family, 2007. <http://www.tilera.com>.
- [23] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.
- [24] Hans P. Zima. Introspection in a Massively Parallel PIM-Based Architecture. In G. R. Joubert, editor, *Advances in Parallel Computing Volume 13*, pages 441–448. Elsevier B.V., Amsterdam, The Netherlands, 2004.
- [25] Hans P. Zima and Barbara M. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series, 1991.