# Self-Stabilization as a Foundation for Autonomic Computing[*]
(Position Paper)

Olga Brukman, Shlomi Dolev, Yinnon Haviv, Reuven Yagel[†]
Department of Computer Science, Ben-Gurion University,
Beer-Sheva, 84105, Israel
{brukman,dolev,haviv,yagel}@cs.bgu.ac.il.

## Abstract

*This position paper advocates the use of the well defined and provable self-stabilization property of a system, to achieve the goals of the self-* paradigms and autonomic computing. Several recent results starting from hardware concerns, continuing with the operating system, and ending in the applications, are integrated: the self-stabilizing microprocessor, with the self-stabilizing operating system, the self-stabilization preserving compiler, and the self-stabilizing autonomic recoverer for applications.*
**Keywords**: *self-stabilizing systems, autonomic computing, liveness, safety, automatic recovery, self-healing.*

## 1 Introduction

The interest in robust systems has increased dramatically in recent years. New terms such as self-healing, self-repair, autonomic computing, automatic recovery, and self-stabilization are used to describe the current computing challenges [5, 6, 14, 22]. The challenge is to create new techniques and technologies for designing software for critical systems that will be able to recover automatically from a failure. The examples of systems that require automatic recovery include flight control systems, web servers that support e-commerce, and on-going core system components such as operating systems.

**Self-Stabilization.** Self-stabilization is an elegant approach to designing fault tolerant systems [6]. A self-stabilizing system is designed to start in any possible configuration where system components (processors, processes, communication links, communication buffers) are in an arbitrary state. The idea is to explore the state space of the system, simply by considering any possible content of memory and proving that from every state the system eventually converges to the desired behavior. In other words, given a specification, we will say that an algorithm is a self-stabilizing algorithm for the given specification if: there exists a set of configurations, called *safe configurations*, that being in one of them ensures that the executions that follow respect the specification. In addition, the set of safe configurations is closed, i.e., being in a safe configuration ensures transition to a safe configuration. Starting from any configuration, a self-stabilizing algorithm must reach a safe configuration in a finite time. Thus the execution sequence may have a (finite) prefix in which the system configurations are not safe, but eventually a safe configuration is reached. Self-stabilizing algorithms can be combined such that the output of the first stabilizing algorithm serves the second stabilizing algorithm as an input to form a single self-stabilizing algorithm. For example, self-stabilizing program that executes on top of a self-stabilizing processor, converges to a safe configuration only after the microprocessor would have converged to its desired behavior.

**Self-Stabilization as a Tool for Autonomic Systems.** We claim that critical systems need to be self-stabilizing. Otherwise, once the type and amount of failures the system was designed to cope with, or the assumptions concerning the interaction (input/output allowed sequences) with the outside environment are violated, the system may enter a state from which it will never recover. The self-stabilization property captures the desire to recover automatically from any (unexpected) state. The self-stabilizing system converges to a legal execution once faults stop occurring. The concept of self-stabilization addresses the automatic failure detection and automatic recovery facet of the

[*]Partially supported by the Lynn and William Frankel Center for Computer Sciences, IBM faculty award, Ministry of Science, Rafael, and the Rita Altura trust chair in computer sciences.
[†]Also at: Rafael 3M, POB 2205, Haifa, Israel.

autonomic computing grand challenge.

The microprocessor may fail to execute the sequence due to *soft errors* (also called single event upset). Soft errors are voltage changes caused by cosmic rays or other disturbances. They can change the output value of a gate in a digital circuit. The error is propagated to the operating system processes and to application level processes, which may cause a system failure or illegal execution by the system. The probability of a soft error increases when the feature size decreases, the voltage decreases and the micro-cycle time is shortened. Some calculations of rates of soft errors were presented in [21]. A high-end router with 10 GB of SRAM and an soft error rate of 600 failures in time per megabit can experience an error every 170 hours. A computer with 2 GB of memory on an airplane at altitude of 1100 kilometers has a soft-error rate of 100,000 failures in time per megabit, which implies a potential error every 5 hours.

Current technology considers soft errors in memory circuits by adding redundancy in the form of error correcting codes. Recent studies have shown that the probability of soft errors in the logic circuit will increase in less than a decade to the current probability for it in memory. Thus, error correcting schemes for the memory circuits only will not be sufficient, as other processor components can be affected as well.

Current designs of microprocessors do not incorporate the self-stabilization property [11]. Even a fully verified processor may fail due to a soft error. Current industrial solution is simply adding some physical protection against radiation, which provides protection only up until certain level of radiation. This is a bold example of the need for a self-stabilizing processor where all combinations of bit values are considered.

An operating system is an essential part of most computer systems. The operating system manages the hardware resources, and forms an abstract (virtual) machine that is convenient to program for by higher level application developers. The operating system running on top of a non-self-stabilizing processor may fail due to soft errors experienced by the processor.

The self-stabilizing algorithm designer assumes that the program is translated into a self-stabilizing machine code and is executed by a processor using operating system services. However, the machine code produced by existing compilers for a self-stabilizing algorithm does not preserve self-stabilization properties [9].

The program execution is truly self-stabilizing only if the underlying execution platform is self-stabilizing, i.e., a processor and an operating system are self-stabilizing and a compiler is stabilization preserving.

We present new concepts for a stack of tools that provide a software platform for executing self-stabilizing programs: a self-stabilizing processor [7], a self-stabilizing operating-system [11, 12, 13, 25], and a self-stabilization preserving compiler [9]. An elegant composition technique of self-stabilizing algorithms [6] is used to show that once the underling microprocessor stabilizes the self-stabilizing operating system (which can be started in any arbitrary state) stabilizes, then the self-stabilizing applications that implement the algorithms stabilize.

We suggest a novel approach for designing a software platform for autonomic computing based on the concept of self-stabilization. The approach is different from the a common "monitor–analyze faults–inject alternations" approach of autonomic computing. Self-stabilization can be applied to support stateless and stateful systems. For stateful systems, the system may record its full state (application state, operating system level variables) into non-volatile memory. Then, the recovery action which will be proven to take place from an arbitrary state will recover the system using the last legal snapshot of the system state. The self-stabilization platform may yield (a tolerable) computation overhead, but it provides an opportunity to create an extremely robust software.

We observe that writing a full self-stabilizing application is very difficult. We know that usually software works correctly after restart, but accumulates bugs and becomes faulty. We present complementary tools – a self-stabilizing autonomic recoverer [3] and recovery oriented programming [2] for such applications. The self-stabilizing autonomic recoverer is a middleware that relies on the existence of a self-stabilizing processor and a self-stabilizing operating-system, and uses restart as a working tool to achieve long periods of legal execution for the application. In the recovery oriented programming approach we suggest to "inject" code snippets into the system code at compile time. The snippets will monitor the important system properties and initiate recovery upon properties unsatisfiability.

Next we detail the concepts used in the design we proposed for each layer. The final outcome of the layers composition is a self-stabilizing infrastructure for executing self-stabilizing applications.

## 2 Self-Stabilizing Microprocessor

A microprocessor execution is *legal* if the processor repeatedly executes fetch-decode-execute cycle. A microprocessor is self-stabilizing if and only if every execution that starts in an arbitrary configuration reaches a *safe configuration* in a finite number of pulses, i.e., reaches the configuration after which its execution is *legal*. In [7] we present a method for verifying the self-stabilization property of an existing processor and a method for adding self-stabilization property to the existing processor.

**Verifying Self-Stabilization.** A microprocessor can be modeled as a finite state automata. The microprocessor repeatedly executes fetch-decode-execute sequence, thus it has cycles in the automata description. The microprocessor automata has a different cycle for each instruction type it is able to execute. Our goal is to prove that every cycle in the microprocessor automata includes a fetch-decode instructions and proper execution of a machine command pointed to by the program counter.

An explicit generation of a microprocessor transition graph is computationally expensive, both in space and in time. We make an abstraction of the transition graph by using only few relevant variables to represent the automata state.

In this work we examine a micro-code controlled processor. A node in the automata is defined by a value of (control variables, e.g.,) the micro-code program counter and a value of the internal micro-registers. Correct values of these variables are essential for the legal execution of the assembler instructions. We use the method described above to verify that the micro-code controlled processor Mic-1 presented in [26] (Chapter 4) is a self-stabilizing microprocessor.

**Adding Self-Stabilization.** One can ensure that a fetch-decode-execute sequence is eventually executed by using an upper bound on the number of clock pulses that may occur between every two successive executions of a fetch-decode-execute sequence. We assume that every processor repeatedly executes a fetch-decode-execute sequence when it is started from a predefined state (e.g., an initial state defined by the manufacturer). Thus, one can use a watchdog circuit that will detect that the processor has not executed a fetch-decode-execute sequence in the last $t$ clock pulses, where $t$ is the given upper bound. In case a timeout takes place, the watchdog resets the microprocessor to the predefined (initial) state.

We design the watchdog circuit to be self-stabilizing so it will be resilient to soft errors. One can implement the watchdog as a counter that is decremented in every clock pulse, using an exact number of bits needed to count the upper bound on the number of pulses between two successive fetches. We assume that the watchdog counter can be initialized to any possible value (due to a soft error), therefore, causing an immature reset of the microprocessor in the worst case [8].

## 3 Self-Stabilizing Operating System

One approach in designing self-stabilizing operating systems is to consider an existing operating system (e.g., Microsoft Windows, Linux) as a black-box and add components to monitor its activity and take actions accordingly, such that automatic recovery is achieved. We call this approach the **black-box** based approach. The other extreme approach is to write a self-stabilizing operating system from scratch. We call this approach the **tailored** solution approach.

**Black Box [11].** We assume that an operating system code is correct, but it may reach a state that was not expected, namely corrupted variables values (due to memory leaks, unexpected IO sequence from the environment, etc.). A primitive solution for achieving fault-free programs is to design the system to repeatedly access a fixed read only memory device (e.g., compact disk) and reload the executable code from it. The reloading procedure is hardwired in ROM and is operated using watchdog and NMI (Non-Maskable Interrupt) mechanisms.

**Tailored Approach.** An operating system kernel usually contains basic mechanisms for managing hardware resources. Classical Von-Neumann machine includes a processor, a memory device and external I/O devices. The tailored operating system is built (like many other systems) as a kernel that manages these three main resources. The usual efficiency concerns which operating systems must address, are augmented with stabilization requirements. In [11] we investigated scheduling issues. In [12] memory management schemes were addressed, and in [13] device drivers are handled.

**Scheduling [11].** The system is composed of various processes which are executing each in turn. The process loading, executing and scheduling part of the operating system usually forms the lowest and the most basic level. Two main requirements of the scheduler are fairness and stabilization preservation. Fairness means that in every infinite execution every running process is guaranteed to get a chance to run. Stabilization preservation means ensuring that the scheduler preserves the self-stabilization property of a process in spite of the fact that other processes are executed as well (e.g., the scheduler ensures that one process will not corrupt the variables of another process).

The scheduler is the key to executing all other processes, therefore, its correct starting and execution must be guaranteed. The watchdog and non-maskable interrupts mechanisms ensure periodically executing the scheduler. Additionally, the state of the scheduler must be validated for correctness. The scheduler uses a process table for scheduling management. This information must be correct in an on-going execution, and must adapt to different scenarios, e.g., starting applications to handle external inputs. Stabilization preservation is achieved by means of monitoring processes and program code restrictions.

**Memory Management [12].** We deal with two important requirements to the tasks of memory management. The first requirement is the *eventual memory hierarchy consistency.* Memory hierarchies and caching are key ideas in memory management. The memory manger must provide eventual consistency of the various memory levels. The second requirement is the *stabilization preservation* requirement. It means that stabilization proof for a single process $p$ is automatically carried to the case of multiprocessing in spite the fact that context switches occur and the fact that the memory is actually shared. Namely, the actions of other processes will not damage the stabilization property of the process $p$.

We suggest three basic design solutions that follow the evolution of memory management techniques. The first approach allocates the entire available memory to the running process, thus, ensuring exclusion of memory access. Since each process switch requires expensive disk operations, this method is inefficient. The second solution partitions the memory among several running processes and exclusive access is achieved through segmentation and stabilization preservation of the segment partitioning algorithm. Both solutions constrain program to reference addresses in the physical memory only (or even in the partition size) and allow static use of memory only. The last solution uses lease based dynamic schemes, in which the application must renew memory leases in order to ensure the correct operation of a self-stabilizing garbage collector. The dynamic memory manager repeatedly checks for memory portions allocated to a process for which the lease expired, and returns every such memory portion to the available memory pool for reallocation.

**Device Drivers [13]**. Device drivers are programs which are practically an essential part of any operating system. They serve as an adaptation layer by managing the various operation and communication details of I/O devices. They also serve as a translation layer providing consistent and more abstract interface for other programs and the hardware device resources (and sometimes they also add extra services not provided by the hardware devices). Device drivers are known to be a major cause of operating system failures [24].

We define two requirements which should be satisfied in order for the protocol between the operating system and an I/O device to be self-stabilizing. The first requirement (the ping-pong requirement) states that in an infinite system execution, in which there are infinitely many I/O requests, the OS driver and the device controller are infinitely often exchanging requests and replies. The second requirement is about progress and it states that eventually every I/O request is executed completely and correctly according to some protocol specification (e.g., the ATA protocol for storage devices). A device driver and device controller can be viewed as a master and a slave working together according to some protocol to achieve their mission. Thus, the device driver acting as a master can check that the slave is following, e.g. the ATA protocol, correctly.

We suggest two solutions. In the first solution the device controller is not required to be self-stabilizing, and the device driver leases some (usually enough) time to the device controller to complete its tasks. In the second solution we relax the timing constraints by assuming that the device controller itself is also self-stabilizing. Therefore, we only need to guarantee that the execution is carried out by both parties according to the protocol. This is achieved by the device driver performing consistency checks according to its current state.

**Tailored OS Implementation [25]**. Prototype implementations for the various parts presented above, using the Intel Pentium processor architecture [16] were composed. For each part mentioned above we implemented the mechanisms for satisfying the needed requirements. We have also provided detailed proofs of the mechanism correctness The implementation is in Assembly language, using the processor opcodes. The methodology we use for building such critical systems is to examine, with extra care, every instruction while assuming an arbitrary initial state. This is achieved by writing the code directly according to the machine semantics (not relying on current compilers to preserve our requirements), together with line by line examination. This style is sometimes tedious, but is essential to demonstrate the way one should ensure the correctness of a program from any arbitrary initial state. Such a method is especially important when dealing with such a basic component as an operating system kernel. Higher level components and applications can then be composed in ways discussed in Section 5.

Our proofs and prototypes show that it is possible to design a self-stabilizing operating system kernel.

## 4 Self-Stabilization Preserving Compiler

Self-stabilizing algorithms are expressed using guarded commands [5] or pseudo-code. Proof of self-stabilizing algorithms requires the examination of the entire state space and is based on the behavior of the program starting from an arbitrary state. Thus the designer prefers a language in which the state space and the program behavior in every state are explicitly defined. The proof of self-stabilization for algorithms stated in general purpose languages such as C++ or Java is complicated because the program context (stack) is hardwired in the language semantics.

The realization of self-stabilizing algorithms requires conversion of high-level descriptions (algorithms) into programs written in a machine language. Unfortunately, existing compilers do not preserve the stabilization property. In fact, the code generated by existing compilers for a simple *for* loop may run practically infinitely when started from an arbitrary state: the loop counter has a corrupted value such that the termination predicate of the loop will never be satisfied [9]. Moreover, the stack used by the compiled self-stabilizing program may become corrupted. The compiled program should be able to recover from such corrupted state.

Another aspect of creating a stabilization preserving compiler concerns the runtime mechanisms, e.g., the heap. The code generated by the stabilization preserving compiler should be able to recover from any arbitrary state of these mechanisms. In order to avoid the need to prove that the machine code produced by the compiler is self-stabilizing we designed a self-stabilization preserving compiler.

In our work [9] we identify languages best suited for describing self-stabilizing algorithms. These are state based languages, such as guarded commands [5], IO automata [19], ASM [17], etc.. Then, we describe a sufficient condition for a compiler to preserve the stabilization property. We say that the generated program *eventually behaves as* the original program if the following condition holds. The generated program may start in an arbitrary configuration and will reach a safe configuration in a bounded number of steps. After reaching the safe configuration the program *eventually behaves as* the original program.

We present our design for a compiler that preserves the *eventually behaves as* property, i.e., it is a self-stabilization preserving compiler. We implemented a self-stabilization preserving compiler that transforms programs written in a language similar to the abstract state machine (ASM) language [17] into IJVM programs [15]. The existence of a self-stabilization preserving compiler allows the designer to focus on the algorithm in its abstract form and leave the machine-level implementation details to the compiler to deal with.

## 5 Self-Stabilizing Autonomic Recoverer for Applications

Complex systems cannot be fully verified as verification of large systems may require an unreasonable amount of time and space. Such systems usually contain flaws – software bugs. The software industry tests software products extensively to eliminate flaws as much as possible. Software is tested by executing a large, but bounded and non-exhaustive, set of input/output with bounded length scenarios starting from a predefined initial state. Faulty, undesired and unplanned behavior may occur due to scenarios that were not tested prior to releasing the software, and may be hard to reproduce.

On the other hand, a consumer of critical systems would like to have a warranty that the system will operate as it should. It is not always enough to be reimbursed when the software does not operate properly. Software malfunctions may cause damage that can outweigh the software cost. Keeping all this in mind, a consumer of a critical system would like to have a warranty that such a system will operate properly. Consumer requirements may be categorized into safety requirements that ensure that nothing bad happens, and liveness requirements that ensure that eventually something good happens [1].

Up to now, the research into self-stabilizing systems and systems that perceive faults as Byzantine behavior has not been able to cope with the fact that software packages contain bugs with very high probability. In particular, both self-stabilization and Byzantine theory limit the number of Byzantine processors that can be handled [10]. Usually software packages function as required for a long period of time after being started from their initial state. This initial correct behavior can be attributed to the testing done by the software manufacturer, which starts the software from the initial state and which considers bounded-length executions. Systems administrators and users occasionally restart such software as a way of coping with failure. In the case of existing large software systems, self-stabilization and Byzantine theory may be well incorporated with restartability. Restarting a program is analogous to rejuvenating of the program code and the program state, while common software rejuvenation mainly releases resources the program is using.

**Our Approach.** We suggest generic self-stabilizing schemes for monitoring processes to ensure customer requirements [3]. The automatic recoverer can be successfully used for applications that have the restartability property, i.e., once restarted they operate correctly for long enough period to do a sufficient work. We present an architecture of a system that monitors and initiates recovery of subsystems automatically in a hierarchical manner. This approach enables us to enforce recovery (e.g., restart) of only one of the subsystems, while the other subsystems are not affected.

Each critical process and each subsystem of critical processes (a group of processes united by a mutual goal) will have a dedicated monitor process. The monitor will check whether the critical process exists and satisfies its specification. Specifications are predicates on the process input/output variables and a history log of recorded process related information accumulated during the process execution. Specifications are classified into liveness and safety specifications. If the monitored critical process crashes, or fails to satisfy one of the requirements, the monitor will initiate some recovery action. A recovery action does not alternate program behavior. A recovery action can be restarting a process, rescheduling, waiting, etc.. The existence of such monitoring processes will be guaranteed by the operating system. The operating system must have some special features to support our system: it has to be self-stabilizing, and must have a dedicated process responsible for the existence of the monitoring processes. This functionality has to be part of the operating systems kernel. Monitoring processes will detect a process state by recording its input/output sequence using new monitoring technologies, such as introspection solutions provided by runtime reflection tools or by means of recording the OS system calls. This approach is also applicable to the legacy off-the-shelf software packages.

Assuming that a monitored process is a finite state machine, our scheme ensures that critical processes and subsystems satisfy their specifications: they are alive and, moreover, progress towards the accomplishment of their mission. The suggested additional layer is thin (a property which will allow a full correctness proof of its code) and is self-stabilizing in order to ensure eventual recovery.

In [3] we have designed an autonomic recoverer, a thin fully-verified self-stabilizing middleware that ensures safety and liveness properties of a monitored system. We consider a monitored system to be a collection of black box components where inter-component dependencies can be expressed by a directed acyclic hierarchical graph. We provide a detailed example with a full correctness proof of an autonomic recoverer usage to ensure the correct execution of the mutual exclusion algorithm for a bounded number of processes. We also present a prototype for a printers server in which the line printer daemon is monitored in order to ensure recovery of printer systems.

In [2] we have also suggested an alternative approach, recovery oriented programming. Recovery oriented programming treats software as a transparent box. Recovery tuples that include important properties to monitor, recovery actions, history log snapshot instructions, etc. as part of the program. The program code is augmented with automatically generated portions of code for event-driven monitoring and some additional processes are created by the framework for external monitoring. The additional code is generated based on the supplied recovery tuples. Their code is generated automatically too. Combined, event-driven monitoring and external monitoring provide full monitoring of safety and liveness even in the presence of transient faults.

## 6 Conclusions

The usage and usefulness of self-stabilizing systems in critical and remote systems, cannot be overemphasized. We have presented a stack of self-stabilizing building blocks (the processor, the operating system, the stabilization preserving compiler) that combined constitute a self-stabilizing software platform. The self-stabilizing software platform is essential for executing self-stabilizing systems. It also can be used as super robust software platform that can be used to run ordinary (not self-stabilizing) programs using autonomic recoverer. This paper demonstrates how self-stabilization can be used to create truly robust systems with very high availability.

## References

[1] B. Alpern and F. B. Schneider. “Defining Liveness”. *Information Processing Letters*, vol. 21(4), pp. 181-185, 1985.

[2] O. Brukman, S. Dolev. “Recovery Oriented Programming”. *8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS’06)*, pp. 152-168, Dallas, Texas, USA, November 2006.

[3] O. Brukman, S. Dolev, and E. Kolodner. “Self-Stabilizing Autonomic Recoverer for Eventual

Byzantine Software". *IEEE International Conference on Software-Science, Technology & Engineering (SwSTE'03)*, pp. 20-29, Herzelia, Israel, 2003.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, D. Engler. "An empirical study of operating systems errors". *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pp. 73-88, Banff, Canada, 2001.

[5] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control". *Communication of the ACM*, vol. 17, pp. 643-644, 1974.

[6] S. Dolev. "Self-stabilization". The MIT press, March 2000.

[7] S. Dolev, Y. Haviv. "Self-Stabilizing Microprocessor - Analyzing and Overcoming Soft-Errors". *17th International Conference on Architecture of Computing Systems (ARCS04)*, pp. 31-46, 2004. Also in *IEEE Trans. Computers*, vol. 55(4), pp. 385-399, 2006.

[8] S. Dolev, Y. Haviv. "Stabilization Enabling Technology". *8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*, pp. 1-15, Dallas, Texas, USA, November 2006.

[9] S. Dolev, and Y. Haviv, M. Sagiv. "Self-Stabilization Preserving Compiler". *7th International Symposium on Self-Stabilizing Systems (SSS'05)*, pp. 81-95, Barcelona, Spain, October 2005.

[10] S. Dolev and J. L. Welch. "Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults". *Proc. of the Second Workshop on Self-Stabilizing Systems (WSS'95)*, pp. 9.1-9.12, 1995. Also in *Proc. of the 14th Annual ACM Symp. on Principles of Distributed Computing (PODC'95)*, pp. 256, 1995, and in *Journal of the ACM*, vol. 51(5), pp. 780-799, September 2004.

[11] S. Dolev and R. Yagel. "Toward Self-Stabilizing Operating Systems". *2nd International Workshop on Self-Adaptable and Autonomic Computing Systems (SAACS'04)*, pp. 684-688, Zaragoza, Spain, 2004.

[12] S. Dolev and R. Yagel. "Memory Management for Self-Stabilizing Operating Systems". *Proceedings of the 7th Symposium on Self Stabilizing Systems*, Barcelona, Spain, October 2005. To appear in *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 2006.

[13] S. Dolev and R. Yagel. "Self-Stabilizing Device Drivers". *8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*, pp. 276-289, Dallas, Texas, USA, November 2006.

[14] IBM. Autonomic computing. `http://www.research.ibm.com/autonomic`, 2001.

[15] IJVM. IJVM Assembly Language Specification. `http://www.ontko.com/mic1/jas.html`, 1999.

[16] Intel. "The IA-32 Intel Architecture Software Developer's Manual". `http://developer.intel.com/design/pentium4/manuals/`, 2006.

[17] Y. Gurevich, B. Rossman, W. Schulte. "Semantic Essence of AsmL". Microsoft Research Technical Report MSR-TR-2004-27, March 2004.

[18] L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem". *ACM Trans. on Programming Languages and Systems*, vol. 4(3), pp. 382-401, 1982.

[19] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[20] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, T. Austin. "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor". *Proceedings 36th Annual International Symposium on Microarchitecture (MICRO)*, 2003.

[21] R. Mastipuram, E. C. Wee. "Soft errors' impact on system reliability". *Voice of Electronics Engineer*, *http://www.edn.com/article/CA454636.html*, 2004

[22] 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, N. Treuhaft. "Recovery Oriented Computing(ROC): Motivation, definition, techniques and case studies". *UC Berkeley Computer Science Technical Report UCB/CSD-02-1175*, Berkeley, CA, March 2002.

[23] MṠwift. "Improving the Reliability of Commodity Operating Systems". *Ph.D. Dissertation, University of Washington*, Seattle, Washington, USA, 2005.

[24] M. M. Swift, B. N. Bershad, H. M. Levy. “Improving the reliability of commodity operating systems”. *ACM Transactions on Computer Systems (TOCS)*, vol. 23(1), pp. 77-110, February 2005.

[25] SOS download page. `http://www.cs.bgu.ac.il/~yagel/sos`, 2006

[26] A. Tanenbaum. “Structured computer organization”. Prentice-Hall, 1984.