International Journal of Applied Engineering Research
ISSN 0973-4562 Volume 10, Number 12 (2015) pp. 30371-30392

http://www.ripublication.com

Parallel Processing For Using A Shared Memory In A Network With The Help of Context Switching Semaphore Using Self-Healing Approach

[1]Dr. Karthik P., [2]Ravikiran B. A., [3]Suresh K., [4]Manu D. K., [5]Gopalakrishna Murthy C. R.

[1]Associate Professor, Dept. of ECE, K. S. School of Engineering & Management, Bangalore.

[2,3,4,5]Assistant Professor, Dept. of ECE, K. S. School of Engineering & Management, Bangalore

Abstract

As thread level parallelism in applications has continued to grow, research in memory sharing in network architecture has continued to expand. Recent calls in industry have led researchers to develop new techniques for writing multithreaded programs for Computer Architectures, which utilize shared memory. First, as more and more applications become multi-threaded, we expect the number of threads executing on a machine outpacing the number of thread contexts that are available for use. Scheduling these threads will eventually grow to become a critical bottleneck for multi-threaded program performance. Since the number of threads to be executed will be significantly larger than the number of contexts, the operating system will require increasingly larger amount of CPU time in Interconnection networks to schedule these threads efficiently. This scheme demonstrates the effect of thread level data dependencies and maintains efficient execution of all threads in the processor. Secondly, a self-healing scheme is designed to share and secure the information of any system at the same time. "Self-healing" techniques ultimately are dependable computing techniques. Specifically, self healing systems have to think for themselves to get inputs on their own, to boot up for backup systems. However, sharing and protection are two contradictory goals. Protection programs may be completely isolated from each other by executing them on separate non-networked computer.

Keywords: Computer Architecture, Data Production, Shared memory, Interconnection network, Thread level Parallelism, self-healing, semaphore.

Introduction

Computers have become pervasive throughout the society by increasing productivity and mobility, as well as by providing new modes of entertainment, and have been used for a wide range of applications. These Single processor chip multiprocessors are becoming the primary building blocks of computer systems. Extracting additional Instruction Level Parallelism (ILP) has been a much researched area, for keeping execution units busy with work and for achieving better performance. Increasingly, efficient communication between execution units or cores improves performance for many-core chips. High performance on-chip communication is necessary to keep cores fed with work. Additionally, performance is more tightly coupled with router delay than to link transmission delay (the reverse of an off-chip interconnect). Bandwidth in on-chip networks is no longer pin-limited and on-chip wires are a readily available commodity. Self-healing mechanisms complement approaches that stop attacks from succeeding, by preventing the injection of code, transfer of control to injected code, or misuse of existing code. Approaches towards automatically defending software systems have typically focused on ways to proactively, or at runtime, protect an application from attack. These proactive approaches include writing the system in a "safe" language, linking the system with "safe" libraries, transforming the program with artificial diversity, or compiling the program with stack integrity checking. The technique of program shepherding validates branch instructions to prevent transfer of control to injected code, and to make sure that calls into native libraries originate from valid sources. Control Flow Integrity (CFI), observes that high-level programming often assumes properties of control flow, not enforced at the machine level [3,4]. The use of CFI enables the efficient implementation of a software shadow call stack with strong protection guarantees. However, such techniques generally focus on integrity protection at the expense of availability. Control flow is often corrupted because input is eventually incorporated into part of an instruction's opcode set as a jump target, or forms part of an argument to a sensitive system call.

Interconnect Network Architectures

The interconnection network design space contains many dimensions along which architects can optimize for power and performance. Topology is among the first considerations for a modest number of cores, buses, rings and crossbars, and provides viable solutions. Shared buses and simple rings do not provide the scalability (bandwidth) needed to meet the communication demands of many-core architectures, while full crossbars are impractical due to their size and wiring requirements. To date, designers have assumed a packet-switched on-chip network as the communication fabric for many-core chips. A large on-chip network such as a mesh or torus, utilizes routers to determine the resource allocation, switching, flow control and routing of messages at each juncture in the network. In on-chip networks, routers consume significant area and power and therefore must be carefully designed. The different kinds of topology are shown in Figure 1.

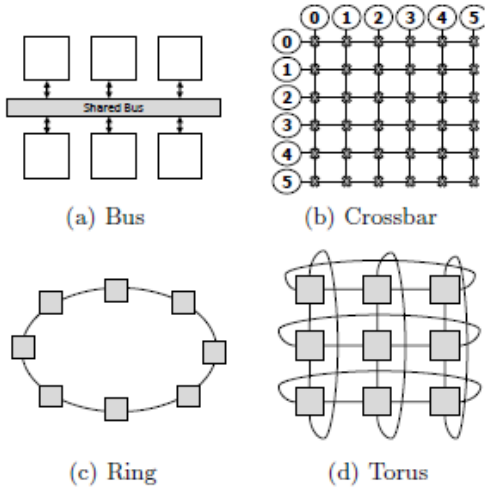

Figure 1: Various Interconnection Networks

In addition, to added area and power overheads of routers results is the increased communication latency over the use of dedicated wires. In order for low-latency, high-throughput communication to be realized, messages from a single source bound for multiple destinations (multicast messages) must also be considered. This requirement to support multicast messages is especially pressing in light of the large number of proposals, and designs that generate multicast messages are finding their way into many-core architectures. In this dissertation, we propose an efficient, low-cost technique to improve router handling of this class of message.

The interconnection network architecture provides the physical resources needed to realize communication. The bits transmitted on that interconnection network architecture are determined by the application's sharing of data and instructions in memory. The shared-memory model is an intuitive way to realize this sharing. The processors access the same memory that allows each processor provides the most perfect (up-to-date) data. The memory hierarchies are employed to improve performance of shared-memory systems. These hierarchies complicate the logical, unified view of memory held in the shared-memory paradigm due to the presence of multiple copies in different caches. Each processor's view of memory must remain consistent with other's view of memory. Cache coherence protocols are designed to maintain one coherent view of memory for all processors. The cache coherence protocol governs what communication is needed in a shared memory multiprocessor to maintain a single coherent view of memory.

Self Healing systems

Self Healing Approach

Self-healing is an approach to detect improper operations of software applications, transactions and business processes, and then to initiate corrective action without disrupting users [21]. Healing systems that require human intervention or intervention

of an agent external to the system can be categorized as assisted-healing systems. The key focus or contrasting idea as compared to dependable systems is that a self-healing system should recover from the abnormal (or unhealthy) state and return to the normative (healthy) state and function as it was, prior to disruption. Some scholars treat self-healing systems as an independent one while others view them as a subclass of traditional fault tolerant computing systems.

The system monitors itself for indications of anomalous behavior. When such behavior is detected, the system enters a self-diagnosis mode that aims to identify the fault and extract as much information as possible with respect to its cause, symptoms, and impact on the system [22]. The system tries to adapt itself by generating candidate fixes, which are tested to find the best target state. Self-healing systems can support decision making in a large way for managerial and organizational situations [25]. Many of the decision support systems (DSS) offer passive forms of decision support, where the decision-making process depends upon the user's initiative. Such active involvement is especially needed in complex decision-making environments.

Architecture of Self-Healing (S-H)

The term ‘self’ in self-healing architecture is referred to the action or response initiated automatically within the system. A general architecture of a self-healing system is shown in Fig. 2.

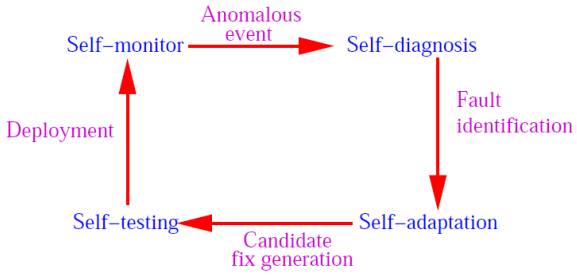

Figure 2: General architecture of a self-healing system

Self Healing Technique

The effective remediation strategies include failure-oblivious computing, error virtualization, rollback of memory updates, and data-structure repair [18]. These approaches may cause a semantically incorrect continuation of execution attempts to address this difficulty by exploring semantically safe alterations of the program’s environment. The technique is subsequently introduced in a modified form as failure-oblivious computing, because the program code is extensively rewritten to include the necessary checks for every memory access the system incurs overheads for a variety of different applications. Data-structure Repair is the most critical concern with recovering from software faults and vulnerability exploits is ensuring the consistency and correctness of program data and state.

Why Self Healing Systems

The software is notoriously buggy and crash-prone, despite considerable work with fault tolerance and reliability [14]. The current approach to ensuring the security and availability of software consists of a mix of different techniques:

a) ***Proactive techniques*****:** seek to make the code as dependable as possible, through a combination of safe languages, libraries and compilers, code analysis tools, formal methods and development methodologies.

b) ***Debugging techniques*****:** aim to make post-fault analysis and recovery as easy as possible for the programmer that is responsible for producing a fix.

c) ***Runtime protection techniques*****:** try to detect the fault using some type of fault isolation, which address specific types of faults or security vulnerabilities.

d) ***Containment techniques*****:** seek to minimize the scope of a successful exploit by isolating the process from the rest of the system, *e.g.*, through use of virtual machine.

e) ***Byzantine fault-tolerance and quorum techniques*****:** rely on redundancy and diversity to create reliable systems out of unreliable components.

Elements of Self-Healing model

In the Self-healing process model, there are different categories of aspects to the self-healing system,

a) ***Fault model*****:** Self-healing systems have the tenets of dependable computing is that called a fault model must be specified for any fault tolerant system. The fault model answers the question of what faults the system is to tolerate. Self-healing systems have a fault model in terms of what injuries (faults), which are expected to be able to self-heal.

b) ***Fault duration*****:** Faults can be permanent, intermittent or transient due to an environmental condition. It is important to state the fault duration assumption of a self-healing approach to understand what situations it addresses.

c) ***Fault manifestation*****:** The severity of the fault manifestation, it affects the system in the absence of a self-healing response. The faults cause immediate system crashes, but, many faults cause less catastrophic consequences, such as system slow-down due to excessive CPU loads, thrashing due to memory hierarchy overloads, resource leakage, file system overflow, and so on.

d) ***Fault source*****:** Thee source of faults can affect self-healing strategies due to implementation defects, requirements defects, operational mistakes, and etc. Self-healing software is designed only to withstand hardware failures such as loss of memory [19] or CPU capacity and not software failures.

e) ***Granularity*****:** The granularity of a failure is the size of the component that is compromised by that fault. Different self-healing mechanisms are probably appropriate depending on the granularity of the failures [20] and hence, the granularity of recovery actions.

f) ***Fault profile expectations*****:** The source of the fault is the profile of fault occurrences that is expected. It considered for self-healing might be only expected faults that is based on design analysis or faults that are unexpected [25]. Additionally, faults might be random and independent, might be

correlated in space or time, or might even be intentional due to malicious intent.

Conventional Methods of Security

The conventional methods can overcome only the effects of passive threats and not the active threats for the authenticate users. They reduce user-friendliness and also, the amount of OS resources required to provide security is high. Different protocol architectures are used for providing security for each layer of the OSI model and it may not be generic. Alternately, in this work, the information is allowed to flow freely through the fetch and decode cycles while an access or authentication is made only between the decode and execute cycle before the data is permanently written into the memory by the user (if authenticated). This is shown in Fig. 3.

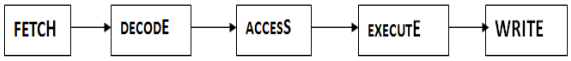

Figure 3: Process of Execution

Proposed features of Security issues

The proposed hardware is shown in Figure 4. The features of the proposed hardware are that it is (i) PCI compliant and (ii) Mounted in a single chip

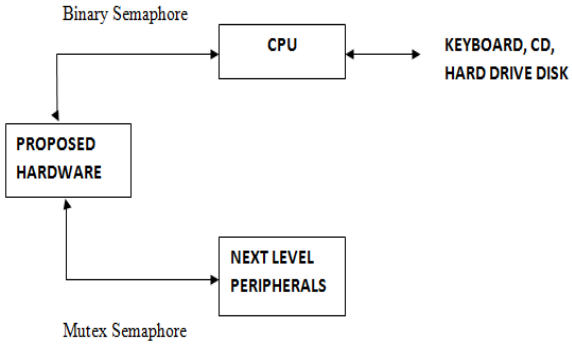

Figure 4: Hardware Process System

a) ***Robustness*****:** It provides defence against vulnerabilities with few false positives or false negatives.
b) ***Flexible*****:** It adapts easily to cover the continuously evolving threats.
c) ***End-to-End*****:** The security policy flows throughout all the seven layers of the OSI model.
d) ***Scalable*****:** It can co-exist with the existing circuitry without any modification [15].

Previous Work

M. Durbhakula, et al., [1998] proposed improving the speed vs. accuracy tradeoff for simulating shared-memory Multiprocessors with ILP Processors. Shared memory multiprocessors are gaining wide popularity as platforms for technical and commercial computing. An architectural simulator for shared-memory systems with processors that aggressively exploit instruction-level parallelism (ILP) requires several hours to simulate a few seconds of real execution time with reasonable accuracy. Theel O., [1996] presented a dynamic coherence protocol for distributed shared memory enforcing high data availability at low costs. As the number of nodes in the system implementing DSM increases, so does the likelihood that the system will experience node failures. For this reason, tolerating node failures becomes essential for parallel applications with large execution times. Constantine Katsinis, [2003] presented models of distributed-shared-memory on an interconnection network for broadcast communication. Due to advances in fiber-optics and VLSI technology, interconnection networks which allow multiple simultaneous broadcasts are becoming feasible. G. B. Adams, et al., [1987] presented a survey and comparison of fault-tolerant multistage interconnection networks. The interconnection network is responsible for fast and reliable communication among the processing nodes in any parallel computer. K. J. Liszka et al., [1997] presented problems with comparing interconnection networks: Is an alligator better than an armadillo. To increase the number of nodes a hypercube can interconnect, the degree of each node has to be incremented by at least one. Thus, to obtain the next larger hypercube, the number of nodes has to be doubled. To alleviate this scalability problem, incomplete hypercubes were introduced, in which any number of nodes can be interconnected. M. Chaudhuri and M. Heinrich [2004] proposed the impact of negative acknowledgments in shared memory scientific applications. An alternative to negatively acknowledge (NACK) requests when the directory is in a pending or busy state are NACK-free directories protocols, this protocol queues pending requests at the home node until they can be serviced. Z. Ding, et al., [2005] proposed switch design to enable predictive multiplexed switching in multiprocessor networks. This behavior can be due to the presence of migratory sharing and producer-consumer relationships, where the processor sharing exhibits temporal locality and is often limited to a small subset of processors. J. Duato, et al., [1996] proposed a high-performance router architecture for interconnection networks. Wave-switching combines circuit-switching and wave-pipelining but in their design, wormhole-routed and circuit-switched data do not interact and have physically separate resources. S. Kaxiras and C. Young [2000] presented coherence communication prediction in shared memory multiprocessors. Instruction-based prediction has been used to accelerate coherence. The directory access off the critical path for distributed shared memory designs. A. Landin, et al., [1991] presented race-free interconnection networks and multiprocessor consistency. The edges in the constraint graph are labeled by the ordering relationship that indicated the program order of a single thread. J. Liu, et al., [2004] proposed interconnect intellectual property for network on chip. The circuit-switching and time-division multiplexing have been used for on-chip networks that provide multicast functionality, but, suffer from the constrained bandwidth of circuit-switching. M. P.

Malumbres, et al., [1996] presented an efficient implementation of tree-based multicast routing for distributed shared memory multiprocessors. The deadlock-free multicast tree routing uses pruning to prevent deadlock in a wormhole-routed network. D. Wentzlaff, et al., [2007] presented on-chip interconnection architecture of the tile processor. The computer industry is shifting from building large, complex single core chips to building simpler cores and placing dozens or hundreds of these simple cores on a single chip.

Shared Memory Process

In this concept, multiple execution contexts sharing a single address space and use multiple programs (MIMD) or more frequently: multiple copies of one program (SPMD). Implicit communication occurs via loads and stores of data in the network. Shared memory is,

(i) no need for messages, communication happens naturally
(ii) Supports irregular, dynamic communication patterns
(iii) Both DLP and Thread Level Parallelism (TLP)
(iv) Complex hardware and simple software
(v) Must create a uniform view of memory

Multiple copies of the same location in memory architecture is shown in Figure 2.

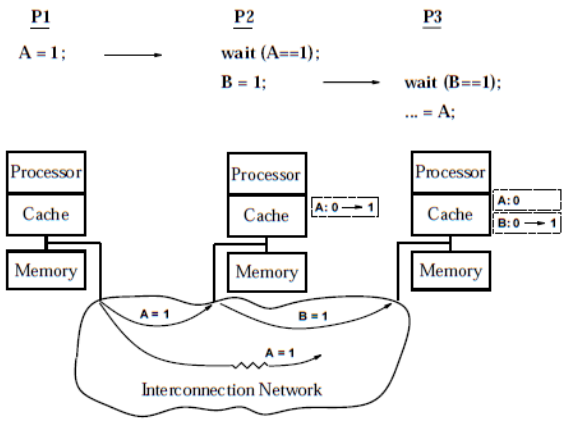

Figure 5: Shared-Memory Multiprocessors

Three aspects to global memory space illusion:

3.1 Coherence

(i) consistent view of individual cache lines
(ii) Absolute coherence not needed, relative coherence OK
(iii) VI and MSI protocols, cache-to-cache transfer optimization

3.2 Synchronization

(i) regulated access to shared data
(ii) Key feature are atomic lock acquisition operation (e.g., t&s)
(iii) Performance optimizations are using test-and-test-and-set, queue locks

3.3 Consistency

(i) consistent global view of all memory locations
(ii) Programmers intuitively expect sequential consistency (SC)
(iii) Global interleaving of individual processor access streams
(iv) Not naturally provided by coherence, needs extra stuff
Order between accesses to different locations becomes important

P1	P2
A = 1;	
Flag = 1;	wait (Flag == 1);
	... = A;

The interconnection network is responsible for fast and reliable communication among the processing nodes in any parallel computer. The demands on the network depend on the parallel computer architecture in which the network is used. Two main parallel computer architectures exist. In the physically shared-memory parallel computer, N processors access M memory modules over an interconnection network as shown in Figure 6(a). In the physically distributed-memory parallel computer, a processor and a memory module form a processor–memory pair that is called processing element (PE). All 'N' PEs are interconnected via an interconnection network as shown in Figure 6(b). In a message-passing system, PEs communicate by sending and receiving single messages, while in a distributed-shared-memory system, the distributed PE memory modules act as a single shared address space in which a processor can access any memory cell. This cell is either be in the memory module local to the processor or be in a different PE that has to be accessed over the interconnection network.

Parallel Computer Machines

Parallel computers can be further divided into SIMD and MIMD machines. In single-instruction-stream multiple-data-stream (SIMD) parallel computers, each processor executes the same instruction stream, which is distributed to all processors from a single control unit. All processors operate synchronously and will also generate messages to be transferred over the network synchronously. Thus, the network in SIMD machines has to support synchronous data transfers. In a multiple-instruction stream multiple-data-stream (MIMD) parallel computer, all processors operate asynchronously on their own instruction streams. The network in MIMD machines therefore has to support asynchronous data transfers. The interconnection network is an essential part of any parallel computer. Only if fast and reliable communication over the network is guaranteed will the parallel system exhibit high performance. Many different interconnection networks for parallel computers have been proposed. The shared memory architecture is shown in Figure 6.

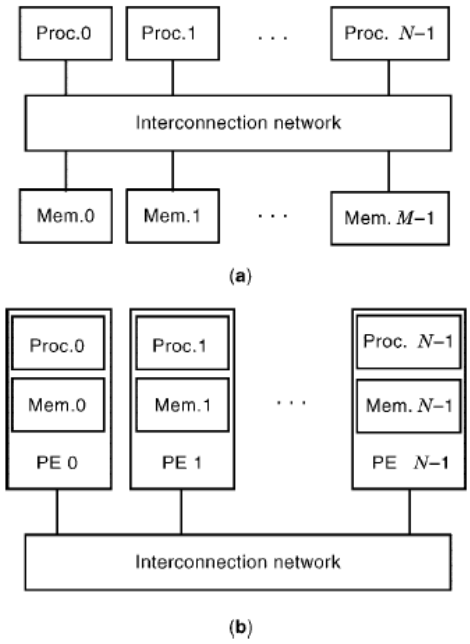

Figure 6: (a) Physically Shared-Memory and (B) Distributed Memory Parallel Computer Architecture

When more than one processor needs to access a memory structure, interconnection networks are needed to route data,

a) from processors to memories (concurrent access to a shared memory structure), or

b) from one PE (processor + memory) to another (to provide a message-passing facility).

Performance Factors

Several metrics are commonly used to describe the performance of interconnection networks:

a) Thread Size: Thread size is a key factor affecting performance in shared memory architecture. Larger threads help increase the scope of parallelism and may help the likelihood of data dependence across threads.

b) Thread dispatch: Thread dispatch plays a major role for fine-grain threads where overhead accounts for a large fraction of thread execution time. Thread completion incurs the overhead of flushing the queues to make memory modifications visible to the system.

c) Load imbalance: A key shortcoming of shared memory architecture is their inability to exploit parallelism in program segments that are not analyzable at compile time. This parameter refers to the degree of dependency on the control flow regularity across threads.

d) Data dependence: Parallelzing structures can often eliminate known data dependence through privatization or reduction optimization. Unknown data dependencies however result in serial program segments which reduces

performance. Using fine grain threads leads to high data dependence and communication across adjacent threads.

Methodology

Semaphores and Resource Sharing

A semaphore is a protected variable or an abstract data type which restricts the access to shared resources such as shared memory in a multiprogramming environment. It is a primitive synchronization mechanism for sharing CPU time and resources. It is a classic solution to prevent race conditions.

Operation of Semaphore

The 'value' of a semaphore is the number of units of resources which are free. To avoid busy-waiting, a semaphore has an associated queue of processes (usually First in First Out). If a process performs a 'P' operation on a semaphore which has the value zero, the process is added to the semaphore's queue. When another process increments the semaphore by performing a 'V' operation, and there are processes on the queue, one of them is removed from the queue and resumes operation.

Binary Semaphore

In binary semaphore, if there is only one resource, the semaphore takes value '0' or '1'.This is explained in Fig. 4.

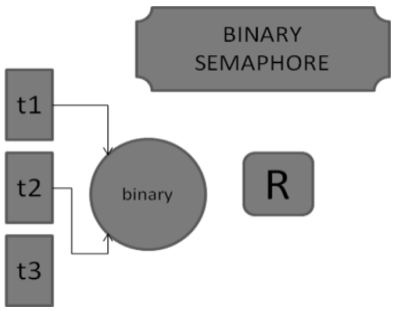

Figure 7: Binary Semaphore

Suppose a 'P' operation busy-waits (uses its turn to do nothing) or maybe sleeps (tells the system not to give it a turn) until a resource is available, where upon it immediately claims one. Now, let 'V' be the operation that simply makes a resource available again after the process has finished using it. The 'P' and 'V' operations must be atomic, i.e., no process may be preempted in the middle of one of those operations to run another operation on the same semaphore. When a semaphore is being used, it takes value '0' and when it takes the value '1', the process directly starts execution without waiting.

Counting Semaphore

The counting semaphore concept can be extended with the ability of claiming or returning more than one unit from the semaphore. When multiple resources are to be shared by many operations, such a semaphore is used. All the resources must be of the same type. This is shown in Fig. 5.

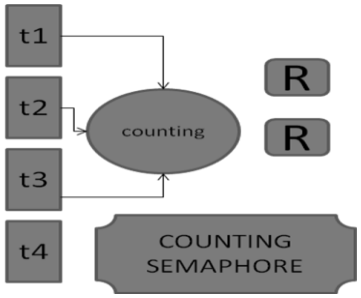

Figure 8: Counting Semaphore

In this, the initial value of semaphore is set equal to number of resources available. As the semaphores are being used, the value keeps decrementing. As the semaphores are being released, after use, its value keeps incrementing. 'Zero' value refers to empty semaphore.

Mutex Semaphore

A mutex is a binary semaphore with extra features like ownership or priority inversion protection. Mutexes are meant to be used for mutual exclusion only. This is shown in Figure 6.

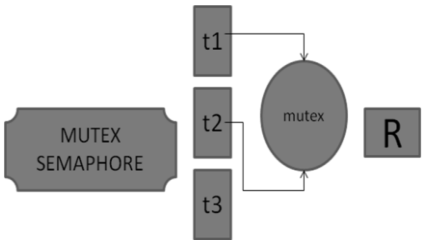

Figure 9: Mutex Semaphore

Initially the semaphore value is set to zero. Once a task attains ownership, it can access the resources as many times as it wants and each time, it accesses, the semaphore value increases.

Characteristics of Semaphore

The characteristics of the three semaphores are shown in TABLE I.

Table 1: Characteristics of Three Semaphores

Binary Semaphore	**Counting Semaphore**	**Mutex Semaphore**
Anyone can release the semaphore. Used for mutual exclusion and event notification.	Only after a task has attained access, it can release the semaphore.	Only the owner can release the semaphore. Used only for mutual exclusion.

Results and Discussion

A configuration consists of eight separate cores with each core four-way multi-thread was considered. This serves the purpose of testing severe workloads with high thread level parallelism corresponds to many requests. This is exactly the status-quo approach to thread scheduling for parallelism implementation. A bus structure that displays read and/or write operations on main memory is used.

An optimum bench mark would consist of several multi-threaded programming applications with more threads than the number of contexts in network architecture. In this scheme SPLASH 2 benchmark is used. They are designed to be multi threaded applications to test networks with several thread contexts. The goal is to utilize the chosen benchmarks to determine the interval time between thread scheduling optimizations that produces optimum performance gain. The FFT kernel and Ocean applications were used to represent SPLASH 2 benchmarks.

The Ocean application and the FFT kernel were appealing as choices for benchmarks since they have very different characteristics in terms of the level of thread level parallelism that they contain. The FFT kernel is one of the programs with the highest amount of thread level parallelism. Ocean on the other hand contains a fair amount of inter-thread data dependencies that enables performance improvement. The amount of thread level parallelism in Ocean is still very small relative to a true multi threaded program compared to FFT. The most important parameters to assess thread level parallelism are which represents the various means of inter thread communication are,

1. Shared Reads
2. Shared Writes
3. Locks.
4. Total Writes
5. Total Reads

Fig. 10 shows the properties of the FFT and Ocean programs.

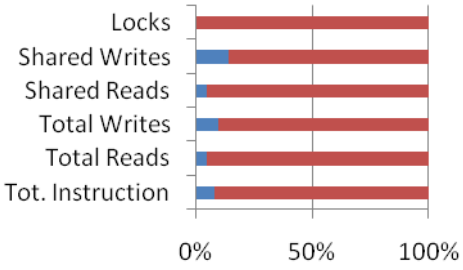

Figure 10: Properties of the FFT and Ocean Programs

The legends used are, --------: FFT, --------: OCEAN

A comparison between performance of FFT and OCEAN applications are shown in Fig. 6. These applications were tested with equivalent work-loads. These comparisons will determine the type of application to be used in parallelism structure. These results shows OCEAN application performance overleaps FFT kernel for thread level parallelism. This leads to the inference that OCEAN application

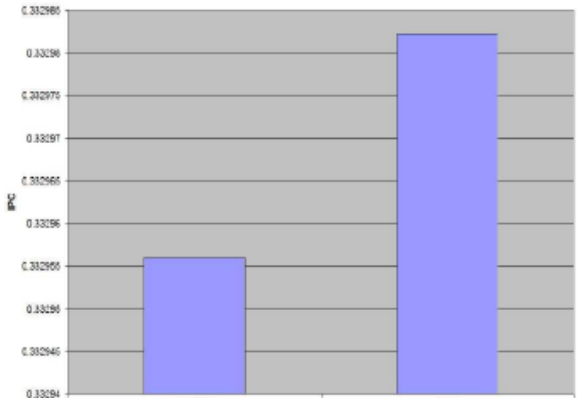

Figure 12: A comparison of The Performance of FFT and OCEAN on Kernel

The effect of varying the time interval between scheduling has on program performance in parallelism structure is shown in Fig. 11 and Fig. 12. The FFT kernel and the Ocean application were run on the network structure. The time interval was varied between from 1 million cycles to 10 million cycles in steps of 3 million cycles. The results of FFT show that a rescheduling interval length of 1 million cycles produced the best Inter Process Communication (IPC). This suggests that using lower interval sizes maximize efficiency. On decreasing the interval size below a certain point leads to a region where the computation time will tend to be larger than the interval length. This would then result in severe problems in network architecture as it would no longer be capable of keeping pace with its scheduling Duties.

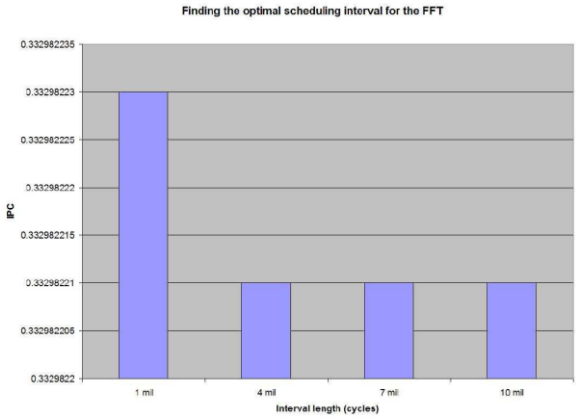

Figure 13: The effect of varying time interval on FFT kernel

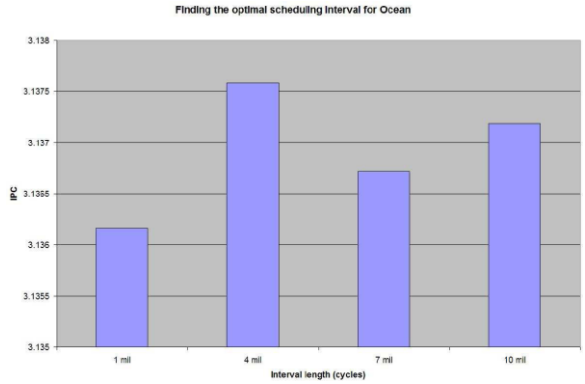

Figure 14: The effect of varying time interval on Ocean kernel

From the results of Ocean it can be inferred that the optimal IPC resulted from choosing an interval size of 4 million cycles. Compared to FFT we see a significant difference in which interval sizes provide the best optimization. It also shows that Ocean contains significantly more thread level dependencies than FFT. Thus it is possible to infer the trend that as thread-level dependencies increase, the size of the optimal scheduling interval will continue to increase. In terms of the performance of this network structure is extremely useful to derive that multi-threaded programs with higher thread-level dependencies, The necessity of choosing an optimal interval size is also demonstrated by these results.

Implementation

In the existing architecture of computers, the bottleneck of connecting a high speed low memory device to a low speed high memory device is solved by using an intermediate memory module called the cache. The cache exists between the high speed CPU and the lower speed memories. However, there is no security between the data link layer of the CPU and the cache. The proposed card is placed on the PCI bus at the maximum possible speed and a direct connection is established to the CPU via

snooping. To understand how the security layer is to be implemented, the knowledge of the three basic terms is required: subject, object and capability. Subject refers to the user or entity which acts on behalf of the user on the system. Objects may be defined as resources within the system. The main term however is capability which is basically a 'token'. The possession of a capability by a subject confers access right for an object. They cannot be easily modified, but they can be reproduced.

The capabilities of the objects are to be stored in the non-readable section of the HDD. For a subject to access a particular object, it must possess the capability for doing so. Hence, before a subject accesses a resource via the CPU, it will first go through a screening check from the hardware on whether or not its capabilities allow it to access such resources. Hence, a security layer is now added to the data link layer between the CPU and the cache. Additionally, user also must be prevented from creating arbitrary capabilities. This can be accomplished by placing the capabilities in special 'Capability Segments' which users cannot access. Another approach is to add a tag bit to each primary storage location. This bit, inaccessible to the user is 'ON' if the location contains a capability. It should be noted that the hardware restricts the manipulation of the location contents to appropriate system routines. If the last remaining capability is destroyed, then that object cannot be used in any manner. In this work, special provisions are made for controlling the copying and movement of capabilities (as well as interpretation) depending on the hardware involved.

Evolved Function is a Semaphore Selector

The schematic of the control block performing the evolved function Fig. 15 Schematic representation of Semaphore Selector. The power consumed by the block under read and write mode is shown in Fig. 8 and Fig. 17 respectively.

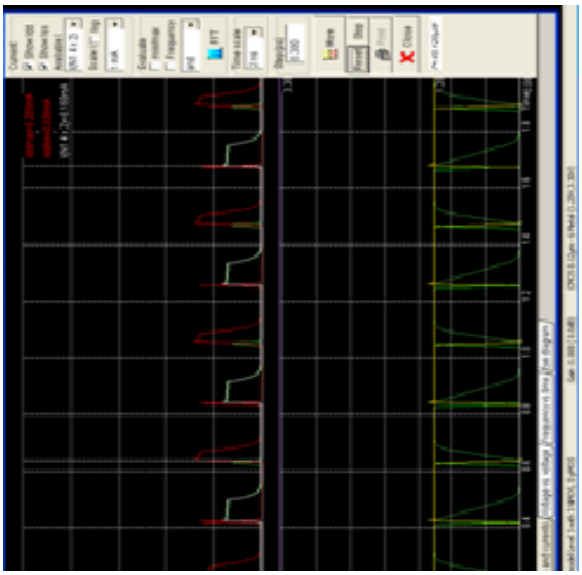

Figure 15: Current and Voltage Variations of Semaphore Selector During Write Cycle

Evolved Function is a Resource Sharing Selector

The control block performing the evolved function of "resource sharing selector" along with its power consumtion is shown in Fig. 18. Similar graph corresponding to read and write cycle is shown in Fig. 11 and Fig. 12 respectively.

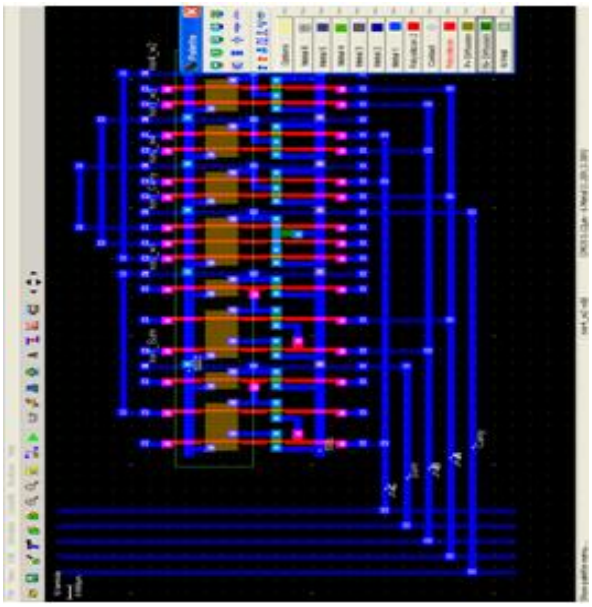

Figure 16: Schematic of Evolved Resource Sharing Selector

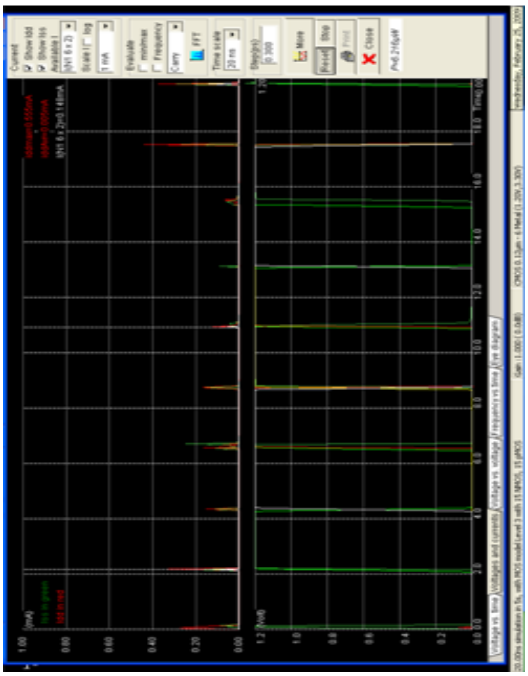

Figure 17: Current and Voltage Variations of evolved resource sharing selector during Read cycle

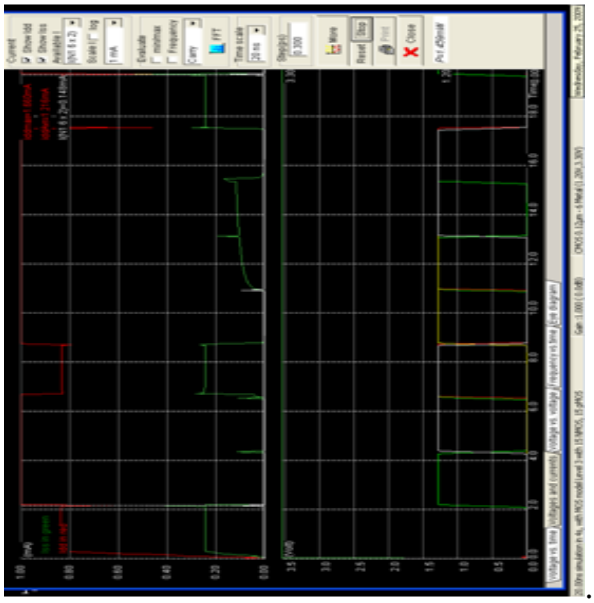

Figure 18: Current and Voltage Variations of evolved resource sharing selector during write cycle.

Evolved Function is a Snoop Selector Function

The schematic of the control block performing the evolved function of a snoop selector circuit is shown in Fig. 21. The power consumed by the PE under read and write cycles is shown in Fig. 22 and Fig. 23 respectively.

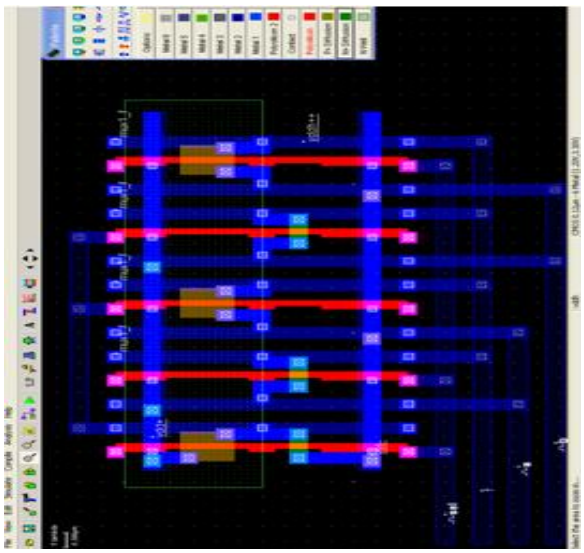

Figure 19: Schematic of Evolved Snoop Selector Circuit

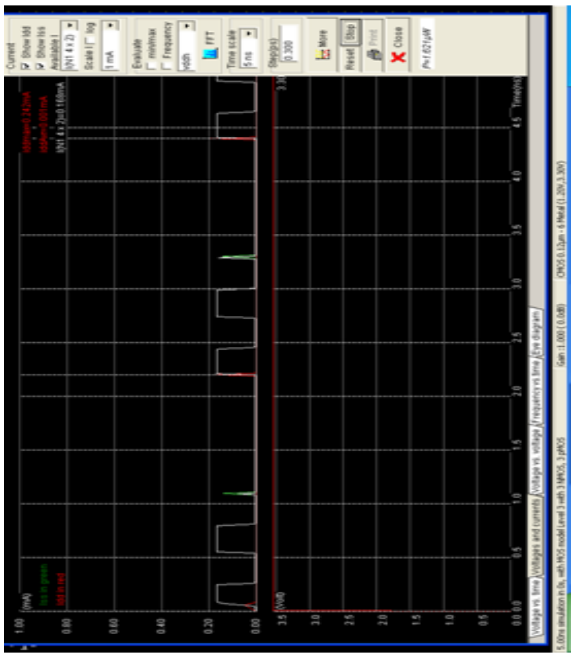

Figure 20: Current and Voltage Variations of Evolved Snoop Selector During Write Cycle

Evolved Function is a Context Switching Semaphore

The schematic of Context switching semaphore block of the evolved function is shown in Fig. 21. The power consumed by the PE during read and write cycle is shown in Fig. 22 and Fig. 23 respectively

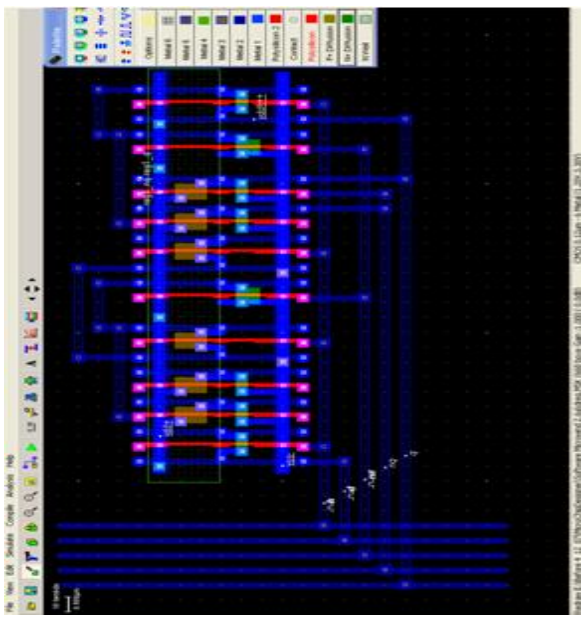

Figure 21: Schematic of Evolved Context Switching Semaphore

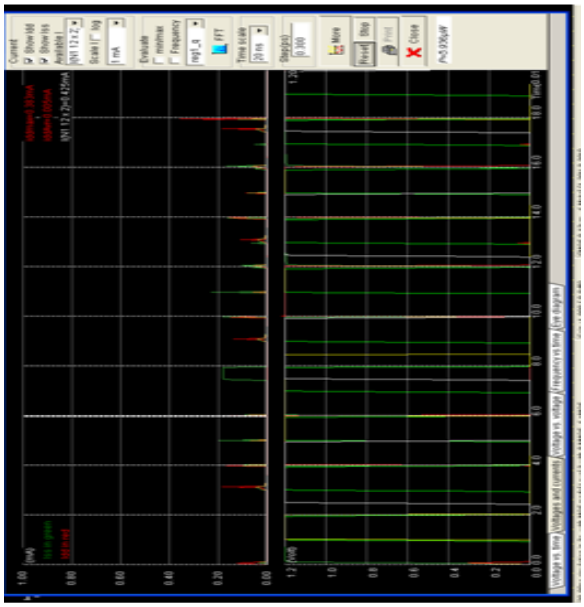

Figure 22: Current and Voltage Variations of evolved Context switching semaphore during read cycle

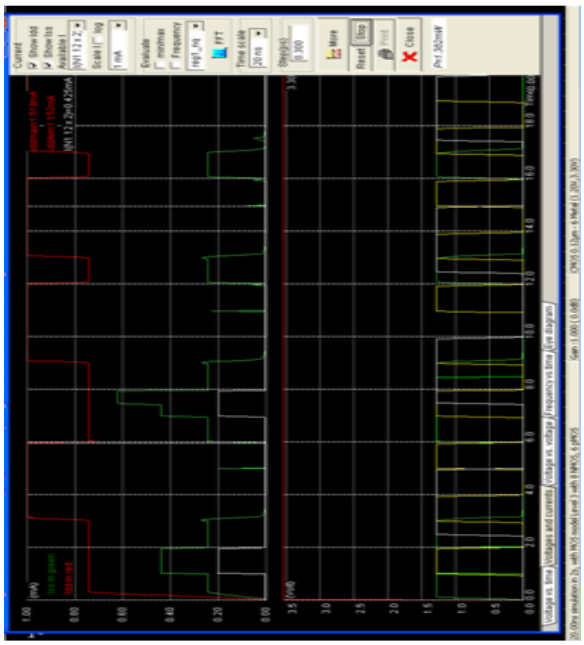

Figure 23: Current and Voltage Variations of evolved Context switching semaphore during write cycle

The results of the discussion are tabulated in TABLE II. It can be inferred from the table that a power level variation exists between the read and write cycles.

Table 2: Power Consumed By Evolved Pe

Evolved Function of PE	**Average Power Consumed by Evolved blocks**	
	read	**write**
Semaphore selector	.3mW	2.325mW
Resource sharing selector	.54mW	3.5mW
Snoop selector	.12mW	3.665mW
Context switching semaphore	.456mW	3.5mW

Self-healing systems prove increasingly important in countering system software based attacks, which recover and secure to the data from interrupted services. Self-healing systems offer an active form of decision support, without human intervention that can detect the fault and recover from the fault. Also, with intelligent architectural models, a self-healing system can select the proper repair plan to deploy the broken component, if there is more than one component that needs to be healed, can prioritize a fault component over the others, etc.

References

[1] M. Durbhakula, V. Pai, and S. Adve, "Improving the Speed vs. Accuracy Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors", Technical Report 9802, Dept. of Elec. and Comp. Engineering, Rice Univ., April 1998.

[2] Theel O., Fleisch B., "A dynamic coherence protocol for distributed shared memory enforcing high data availability at low costs", IEEE Transactions on Parallel and Distributed Systems", vol.7, no.9, p. 915-30, Sept. 1996.

[3] Constantine Katsinis, "Models of Distributed-shared-memory on An Interconnection Network for Broadcast Communication", in Journal of Interconnection Networks, v. 4, n. 1, March 2003.

[4] G. B. Adams III, D. P. Agrawal, and H. J. Siegel, A survey and comparison of fault-tolerant multistage interconnection networks, IEEE Comput., 20 (6): 14–27, 1987.

[5] K. J. Liszka and J. K. Antonio, H. J. Siegel, "Problems with comparing interconnection networks: Is an alligator better than an armadillo", in IEEE Concurrency, 5 (4): 18–28, 1997.

[6] M. Chaudhuri and M. Heinrich, "The impact of negative acknowledgments in shared memory scientific applications", in IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 2, February 2004.

[7] Z. Ding, R. Hoare, A. Jones, D. Li, S. Shao, S. Tung, J. Zheng, and R. Melhem, "Switch design to enable predictive multiplexed switching in multiprocessor networks," in proceedings of 19th IEEE International Parallel and Distributed Processing Symposium, 2005.

[8] J. Duato, P. Lopez, F. Silla, and S. Yalamanchili, "A high-performance router architecture for interconnection networks", in proceedings of the International Conference on Parallel Processing, 1996.

[9] S. Kaxiras and C. Young, "Coherence communication prediction in shared memory multiprocessors," in proceedings of 6th International Symposium on High Performance Computer Architecture, 2000.

[10] A. Landin, E. Hagersten, and S. Haridi, "Race-free interconnection ntworks and multiprocessor consistency", in proceedings of the 18th Annual International Symposium on Computer Architecture, 1991.

[11] J. Liu, L.-R. Zeng, and J. Tenhunen, "Interconnect intellectual property for network on chip", Journal of System Architecture, 2004.

[12] M. P. Malumbres, J. Duato, and J. Torrellas, "An efficient implementation of tree-based multicast routing for distributed shared memory multiprocessors", in proceedings of the IEEE International Symposium on Parallel and Distributed Processing, 1996.

[13] D. Wentzlaff, P. Griffin, H. Hoffman, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor", IEEE Micro, pp. 15–31, 2007.

[14] Bouricius, W.G., Carter, W.C. & Schneider, P.R, "Reliability modeling techniques for self-repairing computer systems", in proceedings of 24th National Conference, ACM, 1969, pp. 395-309.

[15] Shelton, C., Koopman, P.&Nace, W., "A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems", WORDS03, January 2003.

[16] G. Edward Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking", in proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI), October 2004.

[17] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti, "Control-Flow Integrity: Principles, Implementations, and Applications", in proceedings of the ACM Conference on Computer and Communications Security (CCS), 2005.

[18] B. Demsky and M. C. Rinard, "Automatic Detection and Repair of Errors in Data Structures", in proceedings of ACM OOPSLA, October 2003.

[19] M. Locasto, S. Sidiroglou, and A.D Keromytis, "Software Self-Healing Using Collaborative Application Communities", in proceedings of the Internet Society (ISOC) Symposium on Network and Distributed Systems Security (SNDSS), February 2006.

[20] J. Kong, X. Hong, J.-S. Park, Y. Yi, and M. Gerla, "L'Hospital: Self-healing Secure Routing for Mobile Ad-hoc Networks", in Technical Report CSD-TR040055, Dept. of Computer Science, UCLA, January 2005.

[21] Michael E. Shin and Jung Hoon An, "Self-Reconfiguration in Self-Healing Systems", in proceedings of the Third IEEE International Workshop on EASE'06, pp 89-98 (2006).

[22] E. M. Dashofy, A. V. D. Hoek, and R. N. Taylor, "Towards architecture-based self-healing systems", in p*roceedings of the first workshop on Self-healing systems*, Charleston, South Carolina, pp. 21-26, 2002.

[23] H.You, V.Vittal, Z.Yang, "Self-healing in power systems: an approach using islanding and rate of frequency decline based load shedding", IEEE Transaction on Power System, Vol.18, No.1, 2003, pp.174 -181.

[24] T.A. Ramesh Kumaar and Dr.I.A.Chidambaram, "Self-Healing Strategy for Dynamic Security Assessment and Power System Restoration", in International Journal of Computer Science & Emerging Technologies, (E-ISSN: 2044-6004) Vol. 2, Issue 2, April 2011.