An Architectural Framework for Detecting Process Hangs/Crashes

Nithin Nakka, Giacinto Paolo Saggese, Zbigniew Kalbarczyk, and Ravishankar K. Iyer

Center for Reliable and High Performance Computing,
Coordinated Science Laboratory,
University of Illinois at Urbana-Champaign,
1308 West Main St., Urbana IL 61801
{nakka, saggese, kalbar, iyer}@crhc.uiuc.edu

Abstract. This paper addresses the challenges faced in practical implementation of heartbeat-based process/crash and hang detection. We propose an in-processor hardware module to reduce error detection latency and instrumentation overhead. Three hardware techniques integrated into the main pipeline of a superscalar processor are presented. The techniques discussed in this work are: (i) Instruction Count Heartbeat (ICH), which detects process crashes and a class of hangs where the process exists but is not executing any instructions, (ii) Infinite Loop Hang Detector (ILHD), which captures process hangs in infinite execution of legitimate loops, and (iii) Sequential Code Hang Detector (SCHD), which detects process hangs in illegal loops. The proposed design has the following unique features: 1) operating system neutral detection techniques, 2) elimination of any instrumentation for detection of all application crashes and OS hangs, and 3) an automated and light-weight compile-time instrumentation methodology to detect all process hangs (including infinite loops), the detection being performed in the hardware module at runtime. The proposed techniques can support heartbeat protocols to detect operating system/process crashes and hangs in distributed systems. Evaluation of the techniques for hang detection show a low 1.6% performance overhead and 6% memory overhead for the instrumentation. The crash detection technique does not incur any performance overhead and has a latency of a few instructions.

1 Introduction

Usually, a process is said to have *crashed* if it encounters an exceptional condition such that it is no longer able to continue execution. We say a system or process is hung if:

a) it simply does not respond to external inputs, makes no progress, and there is no notification of a failure except as observed by a user or an external monitoring entity, *b)* the process or system goes into an infinite loop[1], detectable only by a dedicated monitor.

The use of heartbeats (HB) to detect failed or hung processes or processors is a common baseline technique [3][18][17], frequently employed in commercial systems. The operating system (OS) can detect most process crashes and detect OS crashes by capturing a wide variety of OS exception conditions and kernel panics, but it is usually unable to detect process hangs and its own hangs. Current approaches detect application/OS hangs by either instrumenting the monitored application to provide an acknowledgment or an "*I am alive"* message, or via a subordinate thread added to the application to enable proactive generation of heartbeats (e.g. Tandem GUARDIAN [26]). The detection of infinite loop conditions is more problematic (without resorting to duplication or self checking mechanisms) since most often the HB thread or the process itself continues to send the "I am alive" message even though the application does not make any progress.

An alternative approach to detect OS crashes/hangs that does not require OS instrumentation uses watchdog timers. A simple user process is employed to periodically reset the watchdog timer, otherwise the OS is considered to be crashed or hung. A limitation of this technique is that an OS crash/hang is indistinguishable from the case where the user-level process that resets the timer abnormally terminates. Another approach used in custom high availability operating systems is to modify the operating system to generate its own heartbeats, again in a separate thread or function [26]. Thus, while it is conceptually simple to assure the existence of a HB mechanism, the design and implementation of a comprehensive HB mechanism is nontrivial. There are three issues which pose significant challenges to HB designers:

Application Intrusiveness. Conventional heartbeat mechanisms detect hangs via augmentation/instrumentation of an application in the form of: (i) *a subordinate thread* – It is possible that the subordinate thread sends heartbeats even if the main process is no longer functional or executes in an infinite loop, or (ii) *progress indicators* (e.g. [21]) – a detailed knowledge of the application and access to the application source code may be required.

Detection Latency. It is essential to have a HB mechanism that detects both process or OS crash/hang with as small a latency as possible. An undetected crash or hang can result in unpredictable behavior of a system and, as a consequence, have a significant negative impact on application or system availability. Since the detection latency depends on a timeout period employed by the HB mechanism, a careful tuning (usually an ad hoc and arbitrary procedure) of the timeout mechanism is required to suit particular application environments and to ensure rapid failure detection.

[1] In our recent study on fault injection based characterization of error sensitivity of Linux kernel on Intel platform (Pentium4 running Linux Kernel 2.4.22), we found that approximately 75% of the manifested errors lead to a crash (easily detectable via heartbeats). Of the remaining 25% of the manifested errors, in about two-thirds of the cases (16%) the processor stopped executing instructions, i.e., processor was frozen. In the remaining one third of the cases (8.3%), the processor continued to execute instructions without doing any useful work e.g., executing in an infinite loop [20].

Determining Timeouts. The problem of determining the timeout values for heartbeats can be addressed using adaptive timeout mechanisms [14][17], although the practical efficiency of these approaches is yet to be determined. Commercial operating systems and distributed or networked environments use an empirical approach to arrive at a fixed timeout value. Timeout values so obtained typically use worst case scenarios, leading to over design and excessive error detection latency. Other solutions used in modern operating systems poll for resource usage [11]; these solutions fail in cases where the process is in an infinite loop without excessively draining any resources. The hardware and performance overhead of self-checking components, as used in the Tandem Guardian system [22] where each of the processes in the process pair heartbeats one another to detect potential hangs, is cost-effective only in high-end server systems (or in switching systems, e.g. AT&T 5ESS Switch [25]).

In overcoming the limitations of the current approaches and addressing the abovementioned challenges, this paper presents a hardware solution in the form of an in-processor hardware module to support low latency crash and hang detection that has the following unique features:

1. It is operating system neutral.
2. It eliminates the need for any instrumentation to detect all OS Hangs.
3. It provides an automated and lightweight compile-time instrumentation methodology to detect all process hangs (including infinite loops), the detection being performed in the hardware module at runtime.

2 Overview of the Approach

We propose in-processor hardware based techniques to provide low latency crash and hang detection with minimal intrusion to application and low hardware overhead. Three hardware modules are introduced:

The Instruction Count Heartbeat (ICH), which uses existing processor-level features (i.e., performance counters) to monitor whether a processor continues to execute instructions in the context of a specific process. This enables detecting abnormal process termination (i.e., process crashes) and also the detection of process/processor hangs where the process still exists but no more instructions are executed. The ICH does this by monitoring whether a fixed number of instructions of the process are executed within a constant time. It exploits existing performance counters in modern processors and *eliminates any OS and application level instrumentation*. While information regarding the abnormal process termination is available at the OS level, it will be seen that the per-process data collected in the ICH has value in the broader context of locally detecting OS hangs.

The Infinite Loop Hang Detector (ILHD) module detects a process hang due to infinite execution in a legitimate loop. In order to enable this, entry and exit points of loops in the application are instrumented via a compiler (i.e., most compilers are geared to detect loops and can be modified to instrument the application at deterministic points that correspond to the entry and exit points of loops). In addition, an application profiling methodology is employed to determine the timeout values for the heartbeats on a per-loop basis, rather than the usual approach of using a fixed timeout for the entire application.

The Sequential Code Hang Detector (SCHD) module is employed to detect process hang due to infinite execution in illegal loops. An illegal loop can occur when the target address of the terminating branch in the basic block is corrupted and, as a result, the control flow can be subverted to an instruction within the block creating an illegal loop. SCHD detects this scenario by maintaining a log of recently committed instructions and searching for a repetition of an instruction sequence.

The *ILHD* and the *SCHD* require lightweight application instrumentation and achieve high accuracy in determining timeouts. Since the technique automatically instruments the application at loop entry and exit points, it does not require knowledge of the source code.

Fig. 1 depicts: (i) the block diagram of the system with the discussed hardware modules, (ii) the inputs to the modules from the processor pipeline, and (iii) operations performed by the modules on receiving the inputs. Fig. 1 also shows details of an implementation of the ICH Module. The ICH contains multiple sets of counters (*Curr Process Cntr i* and *Prev Process Cntr i*), so as to monitor multiple processes simultaneously (each set of counters being associated with a single process). The *Curr Process Cntr* for a process is incremented when the processor completes execution of a fixed number of instructions (as measured by in-processor performance counters) on behalf of that process. The ICH periodically checks if the *Curr Process Cntr* has been incremented since the previous check. After checking for updates the value in the *Curr Process Cntr* is written to the *Prev Process Cntr*, which is used as a reference for the next check. A separate set of counters, (*Curr OS Counter* and *Prev OS Counter*) are allocated for the OS when it is executing a kernel thread or performing some bookkeeping functions. In addition, a *Counter Array Scan* logic is provided to scan all the process counters and to check for updates on any one of them.

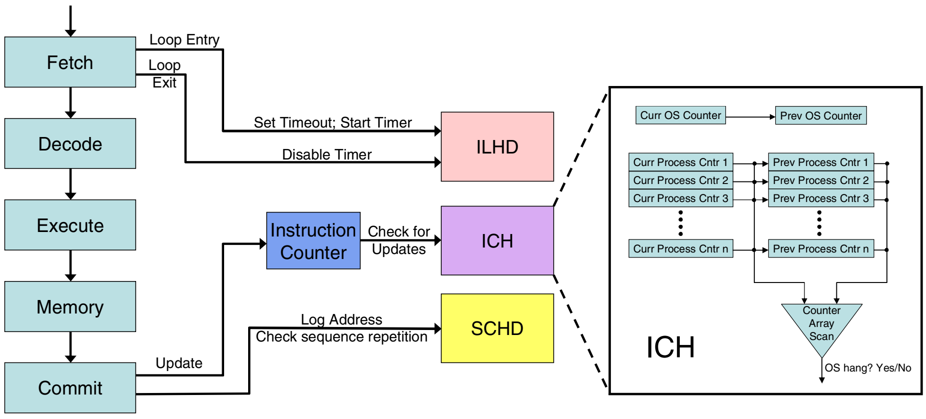


Fig. 1. Block Diagram of Processor with Process Crash/Hang Detection Modules

2.1 Detection of Failure Scenarios

Fig. 2 shows how various failure scenarios can be detected using the techniques introduced in the previous section.

1. Hang when executing user thread
   - *Application Infinite Loop* due to, for instance, design and implementation errors in the application, illegal inputs, or hardware errors that cause the condition for the execution of the loop to be always true – detection is performed by the ILHD or SCHD modules.
   - *System call issued by process never completes* – the detection is performed by the ILHD module, which times out due to the excessive execution time taken by the loop enclosing the system call.

In both cases the module can raise an exception to force the OS to crash/terminate the unresponsive application process.

2. Hang when executing kernel thread.
   - *Processor/Process Frozen* means that the processor/process is not committing any instructions, for instance due to a stuck-at-0 fault on the commit line – the detection is performed by ICH module, which checks the dedicated OS counter to determine lack of progress in executing instructions by the processor.
   - *Kernel Infinite Loop* – the detection can be performed using the *Counter Array Scan* mechanism, which scans all the process counters, at a reasonably long timeout interval, to detect whether the processor stopped executing instructions on behalf of user-level processes. Presence of such conditions indicates that the OS holds on to the processor and the control is never transferred to any user-level process^{2}.

In the above two cases, resulting in a successful detection, a system reboot is initiated.

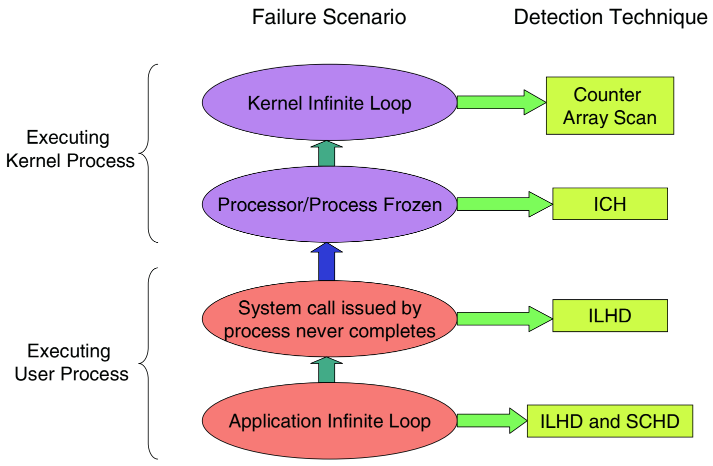


Fig. 2. Hierarchical application of Hang Detection Techniques

^{2} If the interrupts are enabled when the OS is in an infinite loop, the detection can also be done by a technique such as the Kernel Health Monitor (KHM) described in [23]. The KHM is a software implemented timer-interrupt driven monitor to detect OS hangs and to initiate a system reboot on successful detection.

At this point one may ask whether the same goals could be achieved by extending the exception handling mechanism of the operating system with corresponding software implementations of the hardware techniques proposed in this paper. While conceptually this might be possible, there are several disadvantages in taking this approach: 1) each change to the exception handling mechanism is OS specific, whereas the hardware approach is OS neutral, and 2) even if the OS mechanism is used to detect process crashes and hangs, a separate support is still needed to detect OS hangs and crashes. The hardware approach is uniquely suited for achieving this goal

Support for Distributed Systems. While the crash/hang detection has value in the context of a single processor, it can also be leveraged to provide efficient failure detection in a clustered or distributed environment. Fig. 3 shows the logic to provide this support (nodes in Fig. 3 correspond to processors). When a distributed application (consisting of a group of processes) initiates a process *p* on a node *A*, it associates a memory mapped I/O port, *P*, (a location in the node *A*'s memory that can be read by a remote process) with the process and initializes the port. The ICH module in node *A* maintains the information of the *process-port* pair. On detecting a crash of a process *p*, the module raises an exception. The OS handles this exception and writes the id of the crashed process to the corresponding port *P*. Remote processes (members of the initial application process group) can read from this port and, hence, detect the crash of process *p*. In case of an application or OS hang, the hang can be detected and transformed into a crash using the procedure described in Section 0.

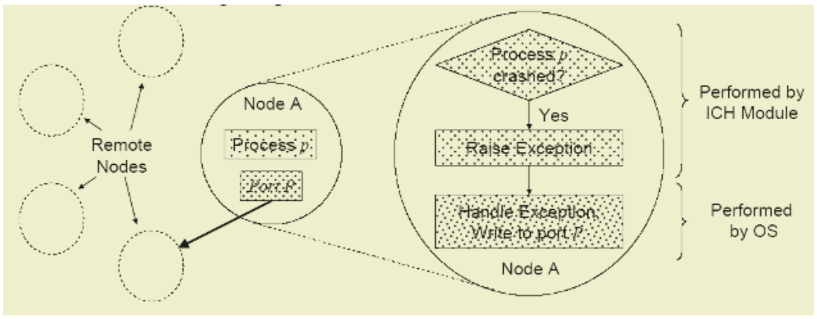


Fig. 3. Crash/Hang detection support for distributed systems

In Summary, the Key Characteristics of the Proposed Approach Are:

- It is hardware implemented and for crash and hang detection requires minimal or no application level instrumentation.
- The crash detection latency can be as small as a few instructions.
- Heartbeats checked concurrently with normal process execution: Since the monitoring entity is a hardware module, the heartbeats from a process *p* are checked concurrently with the execution of *p*. This not only reduces detection latency, but also minimizes the impact of heartbeat on the application performance.

- The scheme provides support in a distributed or networked environment to make failure detectors efficient by *a)* reducing the latency of detection, and *b)* improving the coverage.
- OS neutral: An added advantage of the hardware-based approach over an OS-based one is that it is OS neutral and hence can be leveraged by any OS executing on the hardware.

3 Modifications Needed for the System and Application

All the modules are implemented in hardware and thereby incur hardware overhead. The Infinite Loop Hang Detector (ILHD) and Sequential Code Hang Detector (SCHD) modules require application instrumentation. The aim is to minimize these overheads.

To support the ILHD technique, the application is instrumented with special *CHECK* instructions to notify the monitoring mechanism of the entry and exit points of a loop. Two means of instrumenting the application are explored: (i) use of an *enhanced compiler* – if the application source code is available, a compiler augmented with an additional pass that detects the entry and exit points of loops can be used to embed the CHECK instructions, and (ii) use of a *specialized preprocessor* – if the source code is unavailable, a dedicated preprocessor can be employed to embed the CHECK instructions directly into the application binary (here we leverage our experience using a similar methodology to instrument application binaries with control flow assertion checks [8]). The determination of the timeouts for the loops is done using off-line application profiling. The OS level modifications include support for saving and restoring the state of the modules (timeout values) during a context switch of a process. Fig. 4 shows a flow diagram of the operations to be performed to deploy the ILHD technique.

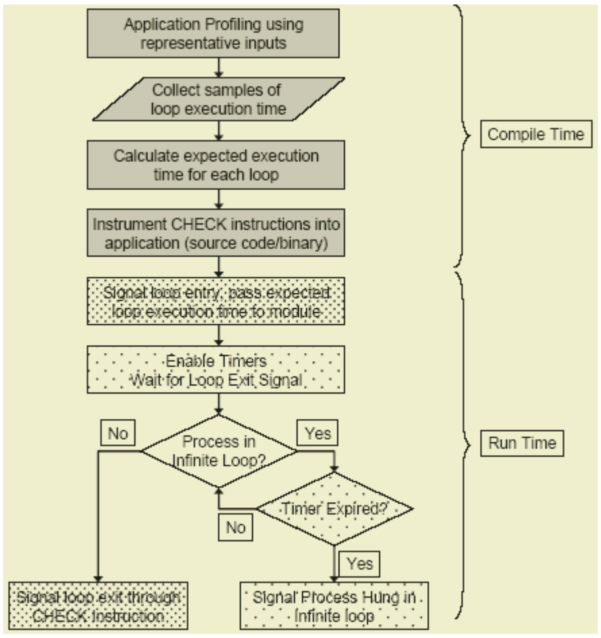
The implementation of the Instruction Count Heartbeat (ICH) module constitutes a set of timers and counters to wait for heartbeats on completion of execution of a fixed number of instructions. Use of multiple counters enables monitoring multiple processes simultaneously in a multiprogramming environment. Also, since there is no feedback from the module to the processor pipeline, the presence of the module does not affect the instruction execution in the pipeline, hence introducing no performance overhead.

The ICH mechanism requires access to performance counters in the processor that count the number of executed instructions. These performance counters need to be part of the process context state. The application can enable the ICH module from the command line by specifying a predefined option (handled by the OS loader during application invocation). Fig. 5 shows the operations needed at application launch time to apply the ICH technique.

The SCHD module requires instrumentation of the application to parameterize the module for the maximum length of illegitimate loops to be checked. This is explained in greater detail in the following section. The instrumentation can be performed using similar techniques as used for the instrumentation for the ILHD module. The SCHD module requires a set of registers, operating as a queue to log the addresses of the committed instructions. It also requires logic to analyze the log and detect a repetition of an instruction. This module has been described in VHDL, implemented on an FPGA, and detailed hardware overhead analysis performed. As shown in Section 0, the SCHD design shows a hardware overhead of about 5%.

4 Description of Modules for Detection of Process Hangs/Crashes

It is assumed that at the processor level we can distinguish between the processes in the system. Current processors provide registers that allow obtaining a pointer to the process descriptor. For instance, the Intel processor running a Linux Kernel maintains this information in the 13 least significant bits of the `esp` register. This can be used to extract the process id of the current process and, thus, distinguish between the processes in the system.


Application Profiling using representative inputs
Collect samples of loop execution time
Calculate expected execution time for each loop
Instrument CHECK instructions into application (source code/binary)
Compile Time
Signal loop entry, pass expected loop execution time to module
Enable Timers Wait for Loop Exit Signal
No
Process in Infinite Loop?
Yes
Timer Expired?
No
Yes
Run Time
Signal loop exit through CHECK instruction
Signal Process Hung in Infinite loop

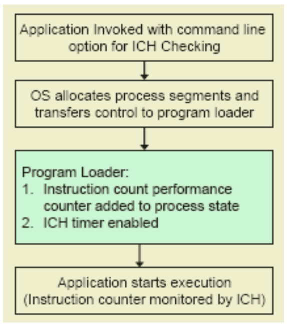

Application Invoked with command line option for ICH Checking
OS allocates process segments and transfers control to program loader
Program Loader:
1. Instruction count performance counter added to process state
2. ICH timer enabled
Application starts execution (Instruction counter monitored by ICH)

Fig. 5. Operations for ICH at process launch

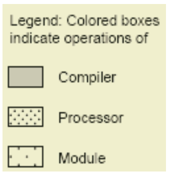


Fig. 4. Flow Chart for Deployment of Infinite Loop Hang Detector Module

4.1 Detecting Process Crash Using an Instruction Count Heartbeat (ICH)

ICH is a generic technique, applicable throughout the application (irrespective of whether it is executing inside or outside a loop), to detect a process hangs using an estimated time for the application to execute a fixed number of instructions. It does not require any instrumentation of the application. A hardware implementation of the module at the processor level allows monitoring the process for progress at the granularity of number of instructions executed. Thus the detection latency for a process crash would be very low, a multiple of the average estimated instruction execution time.

On modern microprocessors like the Pentium and PowerPC, performance registers can be used to count the number of instructions being committed in the processor. On execution of a certain number of instructions, the processor sends a heartbeat to the ICH module by updating a counter. It checks for updates on this counter, and if the counter is not updated before a timeout occurs, the application is declared to be not executing any instructions or in other words, the application is crashed.

Current generation microprocessors (e.g., Pentium) have a built-in mechanism that monitors the commit sequence of the processor. If the processor has not committed any instructions for a certain time, the pipeline is flushed and restarted. However, this mechanism is not application-specific. Using a separate timer for each process, in which the timer and the instruction counter performance register are part of the process state, makes the technique application-specific. ICH can monitor multiple processes simultaneously and does not require modification of the application itself. The only modification required is to associate the process with a timer and to enable the timer. This can be implemented as a wrapper function executed at invocation, requiring no intrusion into the application.

Although this paper describes the crash/hang detection techniques implemented in hardware, the ICH module can be implemented either as a processor-level hardware module or an OS-level software module. The processor updates the Instruction Counter and the Heartbeat Counter on completion of instruction execution. The ICH module, can be implemented either in the hardware or in the OS layers and check the Heartbeat counter periodically for updates.

Timeout for the Heartbeat. The maximum time required by an application to execute a fixed number of instructions depends on the instruction set architecture of the processor. Events such as cache misses or I/O and networking delays are intercepted, and the timer is disabled for the duration of the event.

The following rules determine the timeout policy:

1. If the heartbeat is received before the timer expires, then the timer is reset and execution continues normally.
2. If the timer expires before the heartbeat is received, the module does not immediately declare the process to be hung/crashed. Instead, the module resets the timer to twice the previous timeout value and waits for the heartbeat.
3. On failure to receive the heartbeat for a threshold (t) number of such exponentially backed-off intervals, the module declares the process to be crashed.

Let T be the average estimated time required to execute an instruction of the processor. Then,

The total time waited before declaring a process crash

$$= \tau = \mathsf{T} + 2 \times \mathsf{T} + 2^2 \times \mathsf{T} + \ldots + 2^t \times \mathsf{T} \ = \mathsf{T} \times (2^{t+1} - 1)$$

A limitation of using a timeout of a fixed number of cycles is that the execution time varies widely from one run to another due to cache misses and different instruction execution latencies. To reduce the effect of cache misses, the timer in the module is disabled while a cache-miss is being serviced. In the other cases, a judicious choice of t and T needs to be made. Next we propose a possible hardware implementation for the ICH module.

Hardware Implementation. For each process being monitored, the module contains two registers, *estTime* (T) and *threshold* (t), to store, respectively, the estimated time for the execution of the fixed number of instructions and the threshold for the number of exponentially backed-off intervals before which an application is declared hung. The total time waited τ is calculated using a shift operation on *estTime.* A timer is initialized with this value, and the module waits for the Instruction Counter heartbeat signal from the processor. If the heartbeat is received before the timer expires, then the timer is reset. If the timer expires before the heartbeat is received, the process is assumed to have crashed/hung or entered a deadlocked state.

4.2 Detecting Program Hang Using Infinite Loop Hang Detector (ILHD)

Previous subsections describe hardware-based modules for detecting process crashes that do not require instrumentation of the application. Detection of process hangs on the other hand requires application instrumentation to monitor the application progress. In proposing the Infinite Loop Hang Detector technique we address the two main problems that arise in application instrumentation: (i) placement of instrumented code, (ii) determination of appropriate timeouts. A profiling methodology is employed to monitor the entry and exit points of the loop and derive timeout values on per-loop basis using a profiling methodology. This section describes in detail how the ILHD module achieves these goals.

The problem of finding a program's worst-case execution time is in general undecidable and is equivalent to the halting problem. Significant research work has been done to estimate the loop execution time, since this is a fundamental problem in real time systems. For a program with bounded loops and no recursive or dynamic function calls the loop estimation has been proven to be decidable [9]. Research is in progress for analyzing application with unbounded loops. To detect an application hang, it is assumed that during normal, error-free execution of the program, the execution time for a loop be within a fixed multiple of the execution time in a profile run. If the loop executes for more than this expected time then the application is deemed hung.

Basic Technique. Loop execution time is estimated using static instrumentation to detect loop entry and exit, and dynamic profile information to measure the time elapsed between the arrival of the loop entry and exit signals. During the normal execution of a program, when a loop is entered, a timer is set to the expected execution time of the loop. If the loop is not exited, i.e., the instruction at one of the exit points of the loop is not encountered, before the timer expires, the timer is reset to twice the previous value, waiting for the loop exit condition. If the loop is not exited after a threshold number of such exponentially backed-off intervals, the application is declared to be hung.

Application Instrumentation Through Profiling. In this description, we use terminology for the C programming language to describe loop constructs. The executable dump of the application is statically analyzed to identify all back edges in the program control-flow graph. These are branch or jump instructions with their target address less than or equal to their own address. Each back edge in the program graph is attributed to a loop. There can be more than one exit point for a loop due to a break or a return statement. Each loop is assigned a loop identifier (*loop_id*).

Before each instruction that is the start of a loop or an exit point of the loop, a `CHECK` instruction (`CHECK Loop Start` or `CHECK Loop End`) is inserted to signal the ILHD module of the appropriate event, start of the loop or exit of the loop respectively. Note that a single instruction can be the target for multiple back edges in a program (due to `continue` statements within the loop or due to the loop being executed when one of multiple conditions is true). Similarly, a single instruction may be the exit point for more than one loop.

From the profile run, we obtain samples for the execution times of the loops in the program. The mean (μ_i) and standard deviation (σ_i) for the execution times for loop with *loop_id i* are calculated from this data and stored. Expected execution time for loop i is set to be ($\mu_i + n \times \sigma_i$), where n is a fixed prespecified number to account for the fact that loop execution times for different inputs can be different.

Runtime Monitoring of Loops in the Application. For the normal run of the program, on encountering the start of a loop, the `CHECK Loop Start` signal is sent to the module. The estimated execution time for the loop is sent through an input register in the module. For loops starting at the same instruction, the execution time for the longest loop (the one with the maximum expected execution time) is sent to the module. One drawback of this approach is that it increases the latency of detecting a hang in the loop. On receiving the `CHECK Loop Start` signal, the module sets a timer with the expected execution time for the loop and waits for the `CHECK Loop End` signal. On failure to receive a `CHECK Loop End` signal after a threshold number of exponential backed-off time intervals, the application is declared hung. To handle a set of nested loops, the technique employs multiple timers.

Fig. 6(a) shows the hardware for detecting an application hang in a loop. The module consists of d timers, each associated with a set of two registers, *estTime* and *threshold*, holding the estimated loop execution times and thresholds for the number of exponentially backed-off intervals. Therefore, 2d registers are needed. The *shifter*. block is used to calculate the wait time for the loop, which is stored in the register τ_i. The *Load Timer*$_i$ signal is used to initialize the timer, *Timer*$_i$, with the wait time for the loop. *Start Timer*$_i$ is used to start the timer. Fig. 6(b) shows the finite state machine controlling the d timers. The technique can only check for loops to a nested depth of d. The state of the FSM denotes the nesting depth of the current loop being

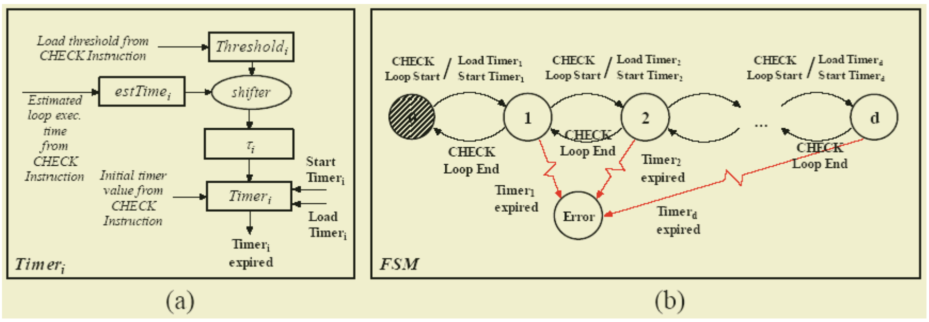

Load threshold from CHECK Instruction
Threshold$_i$
Estimated loop exec. time from CHECK Instruction
estTime$_i$
shifter
τ_i
Initial timer value from CHECK Instruction
Timer$_i$
Start Timer$_i$
Load Timer$_i$
Timer$_i$ expired
Timer$_i$
(a)
CHECK Loop Start / Load Timer$_1$ Start Timer$_1$
CHECK Loop Start / Load Timer$_2$ Start Timer$_2$
CHECK Loop Start / Load Timer$_d$ Start Timer$_d$
0
1
2
...
d
CHECK Loop End
CHECK Loop End
CHECK Loop End
Timer$_1$ expired
Timer$_2$ expired
Timer$_d$ expired
Error
FSM
(b)

Fig. 6. Infinite Loop Hang Detector Hardware

checked for hangs. It is a Mealy machine, whose transitions are triggered by the `CHECK Loop Start` and `CHECK Loop End` signals. The actions taken by the module on receiving these signals is as explained above

4.3 Detecting Program Hang Using Sequential Code Hang Detector (SCHD)

Apart from causing the application to execute in an infinite loop, an error may also lead to the creation of an illegitimate loop created in sequential code of the application. This may possibly lead to a hang. Repeated execution of a sequence of instructions, while the application is executing sequential code, can be used as a trigger for error detection. The SCHD module detects this scenario minimizing the impact on application performance. This section describes different implementations of the SCHD module of progressively increasing complexity while providing a lower overhead in terms of hardware and performance.

To detect repetition of a sequence of instructions, we maintain a log of the previously committed instructions. If a sequence of instructions at the tail of this log of addresses of committed instructions is repeated, then there is a loop. The coverage of the technique or its ability to detect a program hang depends on the number of previously committed instructions being logged. Currently, it is assumed that the processor can commit at most one instruction per clock cycle. This assumption will be relaxed later in the paper. Consider the sequence of instruction addresses where increasing indices refer to instructions issued later in time:

$$\ldots\ a_k,\ a_{k+1},\ \ldots,\ a_{k+L-1},\ a_{k+L},\ a_{k+L+1},\ \ldots,\ a_{k+2L-1}$$

In this sequence, there is a repetition of a sequence of instructions of length L, starting from position k if and only if:

$$a_{k+i} = a_{k+i+L} \qquad \text{for } i = 0, 1, 2, \ldots, L\text{-}1 \tag{1}$$

A queue is maintained to keep track of the addresses of previously committed instructions. Let W be the width of the address expressed in binary format. Let the number of entries in the queue be D. It is necessary that the depth of the queue D be at least $2L$. Letting k = 0, the contents of the queue are:

$b_0,\ b_1,\ \ldots,\ b_{L-1},\ b_L,\ b_{L+1},\ \ldots,\ b_{2L-1}$ where $b_i = a_{k+i}$ for each i ranging from 0 to $2L$-1

b_{2L-1} is the last entry in the queue, b_{2L-2} is the second last entry and so on.

The condition (1) becomes:

$$b_i = b_{i+L} \qquad \text{for } i = 0, 1, 2, \ldots, L\text{-}1 \tag{2}$$

Consequently, the problem of recognizing a repetition becomes simply one of evaluating condition (2).

Efficient Detection of Sequence Repetition in a Single-Issue Processor. In this section, we detail detection of a sequence repetition that improves upon the previous approach by requiring the minimum queue length to be $L+1$ instead of $2L$. The technique described here detects a repetition of a sequence of length L (or a factor of L). Fig. 7 depicts the hardware used to detect the sequence repetition. It shows a queue that contains the addresses of the instructions most recently committed by the pipeline. The number of entries of the queue is $\geq L$+1. The queue is implemented as a shift register. To insert an element at the tail of the queue, all entries of the queue are shifted up by 1 position: (i+1)st entry = i^{th} entry for i = 1, 2, 3, ... , n-1. The incoming element (address of the committed instruction) is placed in the 1st entry.

A comparator is introduced with its 2 inputs connected to the 1st and the (L+1)st entries in the queue. The output of the comparator is '1' if the inputs are equal, '0' otherwise. The output of the comparator drives the increment and reset inputs of a counter. When a new instruction is added to the queue and the output of the comparator is '1', the counter is incremented (which means the instructions at the 1st and (L+1)st entries match). Otherwise, the counter is reset.

Initially the counter is reset to the value '0'. If a sequence of instructions of length L repeats, the counter would be incremented when each instruction of the repeated sequence is added to the queue and hence the counter would be incremented to a value L. When the counter reaches the value L, the repetition of the sequence is noted and the counter is reset.

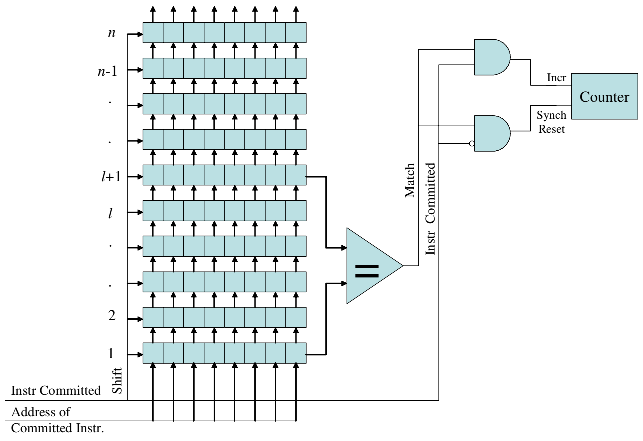


Fig. 7. Basic Sequence Detection Hardware

Detection of Sequence Repetition in a Multiple-Issue Processor. A point to note is that the method described above works only in a single-issue processor. In the case of a superscalar processor that can commit multiple instructions per clock cycle, multiple addresses need to be added to the queue in a single clock, and the pipeline may need to be stalled in order to match the logging of instructions in the module with the committing of instructions in the pipeline. An improved method that can handle multiple committed instructions per clock cycle is presented here. Due to space limitations, we present only the hardware implementation of the technique. Detailed formalization of the technique can be found in [19].

Hardware Architecture of the SCHD Module. We refer to the module detecting a loop of length L as an *L-Loop Searcher* (LLS). Fig. 8 shows the schematic of the *Queue* and of the generic *LLS* module. Let C be the maximum number of instructions that the pipeline can commit in a clock cycle. A SCHD Module detecting any loop whose length is between [L_{min}, L_{max}], requires the following hardware components: (1) a *Queue* module with length (number of entries) $D = L_{max} + C$, (2) ($L_{max}–L_{min}+1$) LLS modules, and (3) an *L'*-input OR-gate, whose inputs are the detect signals, $Detect_L$, coming from the LLS modules. All the modules composing the SCHD Module are clocked by the global clock signal (*clk* in Fig. 8). The enable (*en*), and the reset (*reset*) signals control the memory elements of the design.

The Queue module contains *D* registers, each *W*-bit wide and a Shifter block. The queue can be dynamically configured, through the Shifter block, to enable shifting of *N* positions, where *N* is the number of instructions committed in the current clock cycle by the processor, and ranges from 0 to *C*. The addresses of the committed instructions in a given clock cycle are passed to the queue through the signals in_0, in_1, …, in_{C-1} where the vector $Mask = (Mask_{C-1}, \ldots, Mask_1, Mask_0)$ represents which instructions have been committed. The Shifter block is a multiplexer with *C-1* input ports, each with a width of $D{\times}W$ bits, controlled by the *Mask* signal. It shifts the contents of the queue by *N* positions and appends the instructions that have been committed in the current clock cycle. Note that the output of the shifter is forwarded to the LLS modules in order to detect a loop in the same clock cycle as it occurs. Another possibility is that the output of the Queue component can be the output, *u*, of the register file. This breaks the propagation path of the new instructions from the processor to the SCHD module by using the queue as a buffer. This trades off a shorter clock period with a latency of one clock cycle in the detection of a loop. An LLS module is mainly composed of *C W*-wide comparators, a $log_2(L)$-bit wide register, and some other sparse logic.

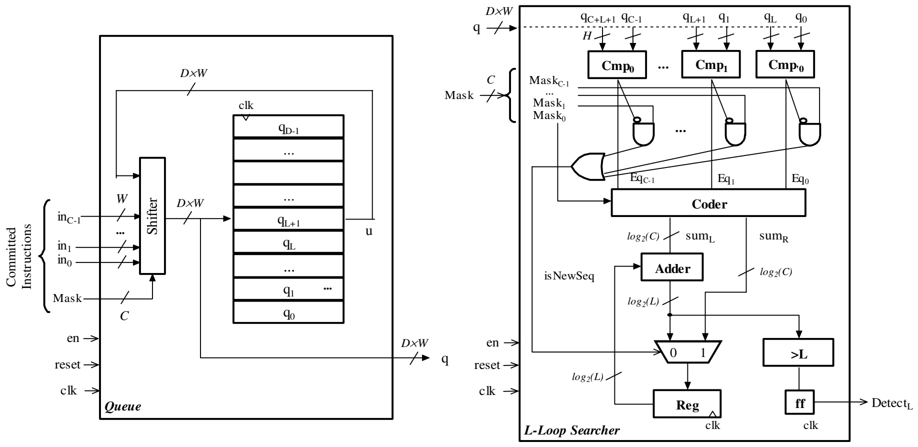


Fig. 8. Hardware Architecture for SCHD Module for Superscalar Processors

5 Implementation and Experimental Results

Two types of experiments have been performed to estimate the overhead in terms of the silicon area for the hardware modules and the overhead for execution time and memory occupation. (1) The ILHD and ICH modules require counters for the timers and registers for holding the execution time parameters. In general, the hardware overhead can be assumed to be negligible, since the modules require simple counters. The SCHD module on the other hand is more complicated, and hence a detailed analysis has been performed. The SCHD module was described in VHDL and synthesized to determine the area overhead. (2) The performance overhead of the ILHD module due to additional `CHECK Loop Start` and `CHECK Loop End` instructions inserted into the instruction stream is calculated in terms of the number of dynamic instructions executed and the execution time.

Area Occupation for the SCHD Module. The area overhead of the SCHD module is evaluated as a percentage of the area required by the processor. In performing this evaluation, we consider a SuperScalar version of the DLX processor [1][7], and implement it using a Xilinx FPGA (VirtexE2000-8) as the target.

The area occupation of the SuperScalar DLX processor is 49% of the overall number of available FPGA slices. The minimum clock period of the synthesized system is about 60 ns. The area occupation of the SCHD is a function of several parameters: (i) the width W of the address of the instructions, (ii) the depth D of the queue, (iii) the width of the loop range $[L_{min}, L_{max}]$ to search for, and (iv) the number C of instructions that the processor can commit in each clock cycle. After the Place-&-Route phase of the design process, we evaluate the area overhead of the Queue, and LLS varying the above mentioned parameters.

We observe that the number of flip-flops required by the Queue module is independent of C, while the LUT count increases, since a more complicated Shifter block able to shift a larger range of positions (0 to C-1) is needed when C increases. Overall, the number of slices required is a small percentage of the overall area available in the device.

The area occupation of the LLS slightly increases as a function of L, with W and C fixed. This is due to the fact that only the Adder, the Comparator, the register and the mux (in the lower part of schematic of the LLS module of Fig. 8) depends logarithmically on L.

The area A_{TOT} of a SCHD module is given by the formula:

$$A_{TOT} = A_{QUEUE}(W, D = L_{max} - C, C) + \sum_{i=L\min}^{L\max} A_{LOOP}(W, i, C)$$

where $A_{QUEUE}(W, D, C)$ and $A_{LOOP}(W, L, C)$ is the area required by the queue and the LLS modules respectively. For instance, the area occupation of a SCHD module checking for loops of length ranging between 4 and 32 instructions, for a processor with $C = 2$, and $W = 8$ is about 424 flip-flops, 904 LUTs, and hence an overall area requirement of 526 slices (since we are looking for repetition of a sequence shorter than 32 instructions, the last 8 bits of the address word are sufficient). The area overhead with respect to a SuperScalar DLX processor (which can commit C = 2 instructions) is about 5%. $L_{max} = 32$ is a reasonable number since the module is used to check for loops shorter than the number of instructions in a basic block and is usually 10-20 instructions. The minimum clock period of the module is less than 15ns, which is much smaller than the minimum period of the checked processor, so the time overhead can be considered negligible.

Performance Evaluation. The heartbeat modules are implemented as an augmentation to the *sim-outorder* simulator of the SimpleScalar Tool Suite [2]. The simulator has been modified to support multiple processes executing in a time-sharing manner. In the experiment multiple processes executing the same application are executed in the simulator. The simulator is modified to provide a heartbeat to the ICH module after a process executes a fixed number of instructions. The application is instrumented with `CHECK Loop Start` and `CHECK Loop End` instructions before each entry and exit of a loop to notify the module of the appropriate event. In the profile mode these signals are used to collect execution time data for the loops and calculate an estimate for execution time for each loop is calculated. The overhead due the CHECK instructions in terms of time taken (in number of cycles) is 0.82% and in terms of the number of instructions executed is 2.93%. Noting that we parameterize the SCHD module (for the length of the loop to be detected) using a `CHECK` instruction at the entry and exit of every loop (when it enters straight line code), we conclude that the overhead due to of the SCHD module would be the same as the overhead of the ILHD module amounting to a combined overhead of is 1.6% and 6% with respect to execution time and memory occupation respectively. The ICH module polls the instruction count performance counter in the processor periodically. It does not provide any feedback to the processor and therefore does not affect the instruction execution in the pipeline, incurring no performance overhead.

6 Related Work

Extensive research has been performed in distributed systems to deal with process hang/crash failures (e.g., [11][12][16]). Starting from Chandra and Toeug's seminal paper [3] on characterizing the properties of failure detectors for solving important problems such as Consensus and Atomic Broadcast, failure detectors have been used in distributed systems to detect process crashes in order to circumvent the Fischer-Lynch-Paterson impossibility result [18]. Felber et al. [10] have classified failure detectors into two categories based on their implementation: *push* and *pull.* Though an object framework for interaction between the different entities has been provided, this work does not address the problem of placement of the heartbeat code in the application code of the objects.

Most previous work, specifically commercially implemented heartbeat mechanisms, have used an empirically derived static value of timeout. For example, in the AIX operating system [11], at fixed intervals, a daemon polls the kernel to check if low priority processes are being starved by higher priority processes. In the Microsoft DCOM Architecture [12], clients send periodic "I am alive" messages to servers. If the message is not received within a fixed amount of time, the client is assumed to have crashed and all its resources are de-allocated. In the Sun HA cluster [13] a hierarchy of heartbeat techniques are used to ensure operation of various entities, (servers, nodes, links in private networks, links with public networks). All these mechanisms use empirically derived timeout mechanisms. Heartbeat protocols using adaptive timeouts try to dynamically adapt to the behavior of the application, the system, and the network. Chen [14], Bertier [15], each improving upon the earlier by a slight modification, propose adaptive timeouts that use a linear combination of the previous *n* arrival times as the current timeout. Being implemented in software, these timeouts do not adapt with the changes in the system very well.

A number of heartbeat protocols have been studied, analyzed, and implemented by Gouda and McGuire [4]. The goals of the approach were to decrease the number of heartbeat messages exchanged and the detection latency. Heartbeats have been implemented in both in software and hardware. The heartbeat mechanism is a standard way of detecting node and application failures in most software-implemented middleware, for example the ARMOR Framework [5]. Murphy [6] gives an informal analysis of the use of watchdog timers for monitoring processes. PCI Bus-based watchdog timers can be used to detect system crash [24]. The watchdog timer is reset by a user process that calls a kernel driver. If the timer is not reset for a configurable period of time the system is rebooted. The watchdog timer cannot distinguish between the failure of the user process and a system crash and can only be associated to a single process and cannot detect crashes/hangs of other processes. Table 1 summarizes some of the related work in this area and discusses their limitations.

Table 1. Summary of Related Heartbeat Work

Technique & Description	Questions/Limitations
AIX [11]: Priority Hang Detection Polling to check lowest priority process Detects starvation of low priority processes	Does not detect incorrect control flow (an error in the branch that tests the exit condition of the loop) that leads to process hang in an infinite loop or process crashes. Uses fixed static timeout chosen by the system designer, either empirically or arbitrarily
Microsoft DCOM Architecture [12]: Pinging mechanism to detect Client crash	Cannot detect a client which is hung. Leads to wasted resources that are allocated to the client. Timeout is fixed at 3 × 120 seconds
PVM [16]: PVM daemons notified of task or host failure, host addition.	Does not detect a process hang.
Chen et al.[14]: Estimates the arrival time for the next heartbeat from the previous *n* heartbeat arrival times. Optimizes detection latency and wrong suspicions	The actual triggers to send heartbeat or to reply to a “Are you alive?” ping message are not dealt with. Timeout mechanism is not application specific.
Accrual failure Detector [17]: Output indicates probability that a process has crashed.	Does not interpret the monitoring information. Other limitations same as Chen’s technique described above.

7 Conclusions

This paper has explored hardware-implemented techniques for detecting application hang/crash. These techniques are implemented as processor-level hardware modules in order to decrease detection latency. The Instruction Count Heartbeat (ICH) module is a non-intrusive technique for detecting process crashes using instruction count performance counters. For detecting processor hangs, we proposed two techniques that require application instrumentation. The Infinite Loop Hang Detector (ILHD) module detects application hangs within a loop by monitoring the execution time of the application in a loop with respect to an estimated value, derived from profiling. It provides a deterministic methodology for instrumenting the application and determining timeout values. The Sequential Code Hang Detector (SCHD) module detects application hangs due to illegal loops created in sequential code due to errors. The CHECK instruction, a special extension of the instruction set architecture of the processor, is the interface of the application with these modules. In hardware on an FPGA device, we implemented the most complex of the modules, the SCHD module, which showed an area overhead of about 5% with respect to a DLX double-issue superscalar processor. Results for the overhead to support both the ILHD and the SCHD modules simultaneously show low execution time overheads of 1.6% and 6% extra memory occupation.

Acknowledgement

This work was supported in part by Gigascale Systems Research Center (GSRC), NSF grants ACI-0121658 ITR/AP and CNS-0406351 ACI-Next Generation Software Prog., and in part by MURI grant N00014-01-1-0576. We thank T. O' Neill for her careful reading of this manuscript.

References

1. H. Eveking, "SuperScalar DLX Documentation," http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/DlxPdf.zip.
2. D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," Tech. Rep. CS-1342, Univ of Wisconsin-Madison, June 1997.
3. T.D. Chandra and S, Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, New York, v.43, n.2, p.225-267, Mar. 1996.
4. M. Gouda, T. McGuire, "Accelerated heartbeat protocols," *Proc. of the Int'l Conf. on Distributed Computing Systems,* May 1998, pp. 202-209.
5. Z. Kalbarczyk, S. Bagchi, K. Whisnant, and R. K. Iyer. "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," *IEEE Trans. on PDS*, 10(6), June 1999.
6. N. Murphy, "Watchdog Timers," *Embedded Systems Programming*, November 2000.
7. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan-Kaufmann, 1996.
8. Z. Yang, "Implementation of Preemptive Control Flow Checking Via Editing of Program Executables," Master's Thesis, University of Illinois at Urbana-Champaign, Dec. 2002.
9. Y.-T. S. Li, et al., "Performance Estimation of Embedded Software with Instruction Cache Modeling", *ACM Trans. on Design Automation of Electronic Systems*, 4(3), pp. 257-279.
10. P. Felber, X. Defago, R. Guerraoui, and P. Oser, "Failure Detectors as First Class Objects," *Proc. of the Int'l Symposium on Distributed Objects and Applications*, 1999.
11. "AIX V 5.1: System Management Concepts," http://publib16.boulder.ibm.com/pseries/en_US/aixbman/admnconc/syshang_intro.htm.
12. G. Eddon, H. Eddon, "Understanding the DCOM Wire Protocol by Analyzing Network Data Packets," *Microsoft Systems Journal,* March 1998.
13. "Sun Cluster 3.1 Concepts Guide," http://docs.sun.com/db/doc/817-0519.
14. W. Chen, S. Toueg, and M. K. Aguilera, "On the Quality of Service of Failure Detectors," *Proc. DSN* 2000.
15. M. Bertier, O. Marin, and P. Sens, "Implementation and Performance Evaluation of an Adaptable Failure Detector," *Proc. DSN 2002.*

16. Geist, A.et.al. *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing,* Scientific and Engineering Series. MIT Press, 1994.
17. N. Hayashibaral, X. Defago, R. Yared, and T. Katayama. "The φ Accrual Failure Detector," IS-RR-2004-010, May 10, 2004.
18. M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM),* 32(2):374–382, 1985.
19. N. Nakka, G.P. Saggese, Z. Kalbarczyk, and R.K. Iyer, "An Architectural Framework for Detecting Process Hangs/Crashes," http://www.crhc.uiuc.edu/~nakka/HCDetect.pdf.
20. W. Gu, Z. Kalbarczyk, and R.K. Iyer, "Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors," *Proc. of DSN 2004,* pp. 827-836.
21. K. Whisnant, R.K. Iyer, Z.T. Kalbarczyk, P.H. Jones III, D.A. Rennels and R. Some, "The Effects of an ARMOR-Based SIFT Environment on the Performance and Dependability of User Applications," IEEE Trans. on Software Engg., 30(4), pp. 257-277, April 2004.
22. Inhwan Lee, R. K. Iyer. "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System," FTCS 1993.
23. D.J. Beauragard. "Error-Injection-Based Failure Profile of the IEEE 1394 Bus," Master's Thesis, University of Illinois at Urbana-Champaign, 2003.
24. "PWDOG1 - PCI Watchdog for Windows XP, 2000, NT, 98, Linux Kernel", http://www.quancom.de/qprod01/homee.htm
25. "AT&T 5ESS™ from top to bottom." http://www.morehouse.org/hin/ess/ess05.htm
26. Daniel P. Siewiorek and Robert S. Swarz. "Reliable Computer Systems: Design and Evaluation", 2nd Edition, Ch. 8.