Concurrent Detection of Software and Hardware Data-Access Faults

Kent D. Wilken, *Member*, *IEEE*, and Timothy Kong

Abstract—A new approach allows low-cost concurrent detection of two important types of faults, software and hardware data-access faults, using an extension of the existing signature monitoring approach. The proposed approach detects data-access faults using a new type of redundant data structure that contains an embedded signature. Low-cost fault detection is achieved using simple architecture support and compiler support that exploit natural redundancies in the data structures, in the instruction set architecture, and in the data-access mechanism. The software data-access faults that the approach can detect include faults that have been shown to cause a high percentage of system failures. Hardware data-access faults that occur in all levels of the data-memory hierarchy are also detectable, including faults in the register file, the data cache, the data-cache TLB, the memory address and data buses, etc. Benchmark results for the MIPS R3000 processor executing code scheduled by a modified GNU C Compiler show that the new approach can concurrently check a high percentage of data accesses, while causing little performance overhead and little memory overhead.

Index Terms—Software fault detection, hardware fault detection, redundant data structure, on-line testing, concurrent error detection, signature monitoring, architecture support for fault detection, compiler support for fault detection.

1 Introduction

Fault detection is among the first actions a reliable computer system must take in response to the failure of a software or hardware component [26]. However, the traditional fault detection approach, duplication, increases the system's cost by somewhat more than 100%. Because many applications requiring fault detection are also cost-sensitive, e.g., embedded systems, low-cost fault detection approaches are needed that provide significant software and/or hardware fault detection.

Fault detection is predicated on redundancy. Traditional fault detection approaches incur the full cost of the redundancy that is used. However, redundancies exist naturally in programs, data structures, instruction set architectures, microarchitectures, computer organizations, etc. Low-cost fault detection is possible if natural redundancies can be exploited for fault detection. This paper proposes a new approach that exploits several natural redundancies to achieve low-cost concurrent detection of two important types of faults, software and hardware data-access faults. The new approach builds on prior work in signature monitoring [19].

Signature monitoring is a general approach that can exploit various natural redundancies to provide low-cost detection of control and control-flow errors. Many researchers, e.g., [21], [24], [25], [29], [34], [35], [37], [38], have proposed signature monitoring techniques that exploit the natural redundancy that control flow is mostly sequential. Signature monitoring uses a variation of block error-detecting codes to encode a program at compile time. The program is later checked during execution by a simple hardware monitor.

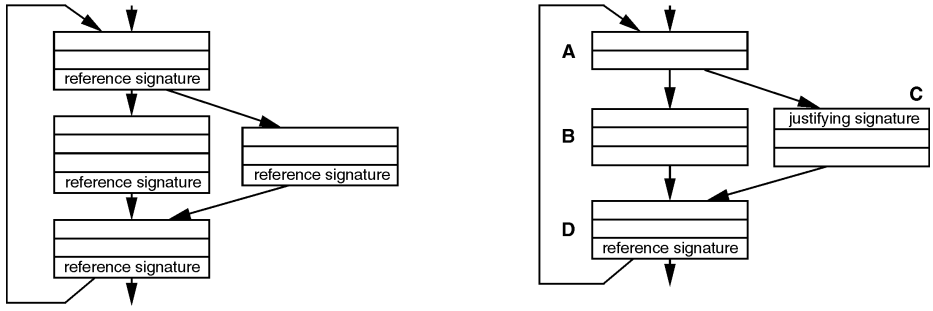
Signature monitoring techniques have been proposed that exploit other natural redundancies. Saxena and McCluskey [24] exploit the redundancy that opcodes have low information entropy, to reduce signature monitoring's error detection latency. Wilken and Shen [38] propose reducing latency by exploiting the redundancy that resources allowed by the instruction set architecture are generally not fully used. Wilken [35] exploits the redundancy that a pipeline is sometimes idle, to reduce performance overhead by placing signatures where idle cycles (pipeline stalls) would otherwise occur. Warter and Hwu [34] and Wilken [35] exploit the redundancy that a program control-flow graph has low information entropy (when viewed as a Markov information source), by placing signatures at low performance-overhead locations.

Beyond detecting control and control-flow errors caused by hardware faults, the existing signature monitoring approach can detect a limited set of software faults. Signature monitoring can detect a software fault that causes a program's binary image to be damaged, e.g., a memory-management software fault that causes an incorrect mapping of a program page. Signature monitoring can also detect a software fault that corrupts a subroutine return address on the stack.

The approach proposed in this paper extends signature monitoring to include low-cost concurrent detection of software data-access faults. The software data-access fault is an important type of fault that has been shown to cause a high percentage of system failures. Sullivan and Chillarege [31] single out software *overlay faults* as a particularly troublesome type of fault. Overlay faults are the subset of software data-access faults that cause errant writes to memory. Sullivan and Chillarege studied field-service software error

• *The authors are with the Department of Electrical and Computer Engineering, University of California, Davis, CA 95616.*
E-mail: {wilken, kong}@ece.ucdavis.edu.

Manuscript received 1 Mar. 1995; revised 5 Dec. 1995.
For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C96331.



(a) Basic Technique. (b) Justifying Signatures.

Fig. 1. Placing program signatures.

reports for the IBM MVS operating system and found that overlay faults caused nearly half of all system failures. They also report that field service personnel consider overlay faults to be particularly hard to find and fix. Miller, Fredriksen, and So [20] injected random inputs into a large set of UNIX utilities to assess the reliability of the utilities. The random inputs caused a large number of system failures. They investigated the cause of each system failure and found that about half of the system failures were caused by software data-access faults. Kao, Iyer, and Tang [16] injected simulated software faults into the UNIX kernel and found that the impact of software data-access faults is higher than the impact of other software faults.

Other prior work discusses the importance of detecting software data-access faults, and proposes software-only approaches for detecting these faults [2], [11]. Austin, Breach, and Sohi [2] use *safe pointers*, pointers that have various attached attributes, to detect software data-access faults. This approach is effective and can be useful during program development for software fault avoidance. However, this approach has an average performance overhead of more than 300%, which as its authors note makes it unsuitable for concurrent detection of software data-access faults [2]. Hasting's and Joyce's `Purify` [11] is a commercially available software tool that provides data-access checking by emulating a two-bit state code for each word in memory. `Purify` also incurs performance overhead that is a few hundreds of percent, and is thus unsuitable for concurrent detection of software faults.

The approach proposed in this paper also provides low-cost concurrent detection of hardware data-access faults. In [10], Gunneflo, Karlsson, and Torin report on processor fault-injection experiments and classify the errors that are caused by the injected faults. The error categories include control and control-flow errors, which can be detected by the existing signature monitoring approach, and data errors, which are not detectable. The experimental results show that a data-access fault occurs with 78% of the data errors. These results suggest that data-access faults are an important type of hardware fault, and that concurrent data-access checking could significantly reduce the number of errors that are not detected by the existing signature monitoring approach.

The remainder of the paper presents the new low-cost approach for concurrent detection of software and hardware data-access faults. For brevity, the approach is presented assuming the target processor is a single-issue RISC processor. Section 2 provides background on signature monitoring related to the new approach. Section 3 introduces the new redundant data structure that allows detection of data-access faults, and presents two architecture-support techniques that allow data accesses to be checked with low performance overhead. Section 4 proposes a compiler instruction-scheduling technique that further reduces performance overhead, and Section 5 presents experimental results. The final section draws conclusions and summarizes the paper's contributions.

2 Signature Monitoring

This section reviews the existing signature monitoring approach and shows how signature monitoring can be applied to a RISC processor, as background for the new data-access checking approach that is presented in the subsequent sections.

2.1 Signaturing a Program

Signature monitoring uses a variation of block error-detecting codes to detect errors [19]. Fig. 1a shows the basic technique, where the compiler places a *reference signature instruction* at the end of each basic block. The reference signature instruction includes an *s*-bit field the compiler sets so that the machine code of the basic block, including its reference signature instruction, is a codeword of the *s*-bit signature function. During program execution, a simple hardware monitor regenerates the signature and checks it at each reference signature. The monitor detects non-codeword signatures that program-memory errors and program control-flow errors produce.

Justifying signatures [21] allow signature monitoring overhead to be reduced. A justifying signature is an *s*-bit field in a *justifying signature instruction* that the compiler sets so that the signatures of two paths are consistent at the location where the paths merge. For Fig. 1b, the compiler sets the justifying signature in basic block C to produce the same signature at block D's first location along path AC as is produced along path AB. The justifying signature allows block D's reference signature to form a codeword for path ABD and for path ACD. This results in half the static signature-instruction count compared with the basic technique, for this example.

Signature consistency must also be ensured where pro-

gram paths merge at the entry to a subroutine, and at the entry to the exception handler. The compiler places a justifying signature along all but one of the paths that lead to a subroutine, to ensure consistency at the subroutine's entry [37]. The signature along the remaining path determines the signature at the subroutine's entry location. The signature of the exception handler's entry location is a predetermined value, e.g., 0. When an exception occurs, the current signature is saved in a frame on the stack along with the rest of the process state [24]. Hardware then resets the signature register to the predetermined value, and a new signature monitoring process starts for the exception handler. Upon return from the exception, the previous signature is restored from the stack, and the previous signature-monitoring process resumes. If the exception handler emulates an instruction that causes an exception, the exception handler must also emulate the signature computation at the instruction [7].

Justifying signatures increase error detection latency because they increase the distance between reference signatures, where signatures are checked. To avoid unbounded error detection latency, a reference signature must be placed inside each inner loop [21]. The compiler can place additional reference signatures based on the application's error-detection-latency requirements. Alternatively, *horizontal reference signatures* [37], one or more reference signature bits included with each instruction, can be used to provide short error detection latency and to eliminate the performance overhead caused by reference signature instructions.

The compiler can place justifying signatures so that performance overhead is reduced far beyond the reduction in static signature-instruction count [35]. For example, the compiler can place a justifying signature instruction in either block B or in block C of Fig. 1b, to produce equal signatures along paths ABD and ACD. If an idle cycle occurs in either block B or block C, the compiler can place the justifying signature instruction at the idle cycle, resulting in zero performance overhead. An efficient graph-construction algorithm produces an optimal placement of justifying signatures, reducing average performance overhead to a fraction of a percent [35].

2.2 Signature Monitoring for a RISC Architecture

A set of simple extensions to a typical RISC instruction set architecture is described that supports the existing signature monitoring approach. An implementation of an on-chip signature monitor for a corresponding pipelined RISC processor is also described. An on-chip monitor is necessary for modern processors, which have on-chip caches, and an on-chip monitor is much less complex than an off-chip monitor [25]. The design of an off-chip monitor for a pipelined RISC processor is presented in [7].

The extended architecture must support reference signature instructions and justifying signature instructions for normal operation, and must support signature saves and signature restores for context switching. The reference signature instruction can be implemented as OP r_0, r_x, `immediate`, where OP is a specific existing opcode, the destination register is r_0 (which is hardwired to the value zero), and the reference signature is the literal in the fields that normally specify the source register r_x and the immediate data. Similarly, the justifying signature instruction can be implemented using OP* r_0, r_x, `immediate`, where OP* is any existing opcode with immediate data, other than OP. These signature-instruction formats allow a simplified implementation because they do not require changes to the processor's instruction decoder.

Two new instructions are used for saving and restoring the signature around a context switch. When an exception occurs, the exception handler saves the current signature using MOVSIG r_x. MOVSIG moves the value from the signature register to the destination general-purpose register r_x. The exception handler then saves the signature to the stack using a conventional STORE. RISC architectures include similar move instructions which transfer certain elements of the process state to a general-purpose register, e.g., [15], [27].

To allow signature monitoring of the execution handler, the monitor includes two registers, the *signature register* and the *shadow signature register*. When an exception occurs, the current signature is preserved in the shadow signature register, the signature register is set to the predetermined value (e.g., 0), and signature monitoring of the exception handler begins. Thus, specifically, MOVSIG moves the signature from the shadow signature register to the destination general-purpose register.

Architecture support for restoring the previous signature at the end of an exception differs depending on whether the architecture includes *delayed branching* [12] or not. For brevity, we only consider the case without delayed branching, which is the approach used by recent RISC architectures, e.g., [13], [27]. A Modified Return From Exception instruction (MRFE) is used to restore the previous signature. A conventional Return From Exception (RFE) is a type of register indirect jump, i.e., RFE r_x, where r_x specifies the return address. To support signature monitoring, MRFE has two operands, the return address and the return signature, i.e., MRFE r_x, r_y. When executed, MRFE transfers the return address from r_x to the program counter in the usual manner, and also transfers the return signature from r_y to the signature register, allowing execution and signature monitoring of the interrupted process to resume.

Fig. 2 shows the datapath for a single-issue RISC processor with a typical five-stage pipeline [12], and the datapath for the corresponding signature monitor. The multiple-input signature register (MISR) that computes the runtime signature is located in the monitor pipeline stage that corresponds with the processor's execution stage. The operand specifier bits of an instruction at this stage are available to the MISR from the operand latch in the processor's pipeline. The remaining bits (the opcode bits) are passed to the MISR from the prior pipeline stage by a small, e.g. 6-bit, latch in the monitor. The first set of XOR gates compresses the complete instruction down to the width of the signature. The compressed result is then XORed with the shifted signature from the signature register, and the resulting signature is stored back to the signature register. At the same time, the unshifted signature and the reference-signature field are compared, and the comparison result is latched if the current instruction is a reference signature instruction.

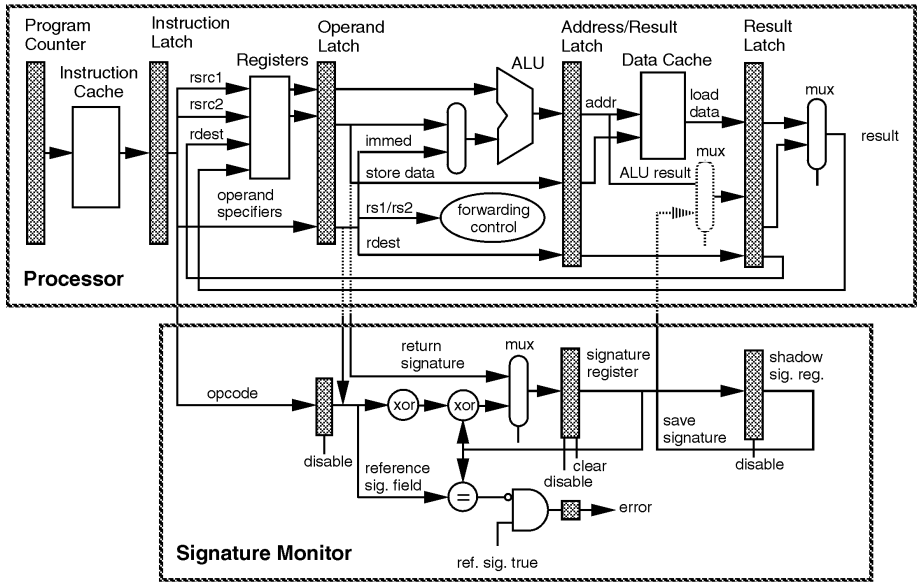


Fig. 2. Datapaths for a pipelined RISC processor and its signature monitor.

To support exceptions, when an instruction advances to the pipeline's write-back stage, the signature that includes the instruction is moved into the shadow signature register. Thus, when an exception occurs, the signature in the shadow signature register includes all completed instructions. For example, when a page-fault exception occurs for an instruction in the data-access stage, the signature in the shadow signature register will include the instruction that just completed the write-back stage. When an instruction stalls in the processor's pipeline, a corresponding stall occurs in the monitor's pipeline so that an instruction and its signature remain synchronized. (A stall is produced by a control signal that disables the clock to the corresponding pipeline latch, as illustrated in the monitor pipeline.)

Signature saves and signature restores are facilitated by a multiplexer in the processor's datapath and a multiplexer in the monitor's datapath, respectively. When an exception occurs, the signature register is cleared, and the clock to the shadow signature register is disabled until MOVSIG transfers the signature through the multiplexer in the processor's data-access stage to the result latch in the processor's pipeline. In the next clock cycle, the signature is written to the destination register specified by MOVSIG. At the end of exception handling, the previous signature is transferred to the signature register through the multiplexer in the monitor when MRFE reaches the execution stage.

As part of a separate but related project, our group implemented a five-stage RISC processor in register-transfer-level VHDL and a corresponding signature monitor in structural VHDL that we synthesized into a CMOS implementation [36]. Using efficient realizations of an XOR gate [33] and a pipeline latch [9], the signature monitor requires only 1,304 transistors, including support for exception handling, context switching, and pipeline stalls. The hardware complexity of the monitor is a fraction of a percent of the complexity of a RISC processor, which typically has several hundred thousand transistors or more.

3 Data-Structure Signatures

A new technique is proposed that embeds a signature into each instance of a data structure, allowing efficient detection of software and hardware data-access faults.

3.1 New Redundant Data Structure

A data structure has natural redundancy in the form of spatial locality because its fields are contiguous. A new technique exploits this redundancy to achieve efficient data-access fault detection. Fig. 3a shows a simple data structure containing no explicit redundancy, along with an illustration of a pointer to the data structure. The proposed technique creates redundancy in a data structure by adding a one-word *data structure signature* (DSS) to each data structure instance. A DSS is a constant that is set when the data structure instance is created, as described below. The DSS is placed in the word that precedes the first element of the data structure. This allows the program to access the elements of the data structure in the usual manner, maintaining language compatibility (e.g., a pointer to the data structure still points to the first element). Fig. 3b shows an example redundant data structure, along with an illustration of a pointer to the data structure. A DSS allows spatial locality to be exploited for data-access fault detection as follows. For a pointer p that under fault-free operation points to the data structure, the compiler knows the offset from the pointer to the DSS is -1 (we assume memory is word addressable, to simplify the presentation). Thus, the compiler can insert instructions that check the pointer p by loading the value at location $p - 1$, and comparing that value with the reference signature for that data structure. If a software or hardware fault damages the pointer, or a hardware fault occurs when the processor accesses the DSS, a check of the runtime DSS will detect the fault. The data structure's spatial locality allows efficient data-access fault detection in part because any access through the pointer to any field in the data structure can be checked using one redundant word.

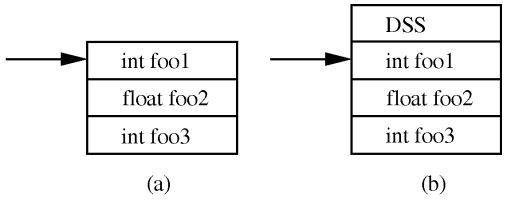


Fig. 3. Example data structure and redundant data structure.

A data-structure signature can be viewed as a type ID assigned by the compiler that is the same for all instances of the data structure. When an instance is created, space is allocated for the DSS and the DSS is set to the assigned ID. The DSS check ensures that the data structure accessed at runtime is the same type that the compiler assumed the corresponding LOAD/STORE would access. In a more refined view, the DSS is a type ID that is used only within the scope of a data structure instance or group of related instances, e.g., within a procedure for a local structure-variable.

Data-structure signatures can be used to check accesses through a pointer to any type of data structure, including a record, a stack frame, an array, etc. Data-structure signatures can also be used to check accesses to a scalar variable by transforming the scalar variable into a structure that has the scalar variable as its only field.

For existing architectures the DSS can be checked before each data access using a four instruction sequence:

```
Load Rx <- [p-1]                /* get DSS */
Load Ry <- ref_sig              /* get reference
                                   sig. */
Compare Rx,Ry                   /* compare DSS,
                                   ref. sig. */
Branch_Not_Equal error handler
Load (Store) Rz <- (->) [p]     /* OK, do usual
                                   access */
```

Note that a reference-signature LOAD inside a loop is *loop invariant* [1], and can be moved outside the loop. This reduces the dynamic length of the checking sequence to about three instructions for frequently executed loops.

The data in [4] suggest that LOADs and STOREs average about 25% of the instructions executed by RISC processors. Thus, checking data accesses in this manner will result in about 75% performance overhead.

3.2 Architecture Support for DSS Checking

Two simple architecture-support techniques are proposed that significantly reduce the performance overhead and instruction-memory overhead for checking data-structure signatures. A compiler instruction-scheduling algorithm proposed in Section 4 uses the two techniques in combination. First, Load Data-Structure Signature (LOAD DSS) is added to the instruction set, with the form: LOAD $\mathtt{r_0}$, $\mathtt{r_{base}}$, `offset`. The compiler sets the offset field to –1. LOAD DSS can be implemented using an existing LOAD opcode, as the special case where the result is written to $\mathtt{r_0}$. This allows a simplified implementation because the processor's instruction decoder does not have to change. For LOAD DSS, the data-structure signature at the effective address is added to the signature, along with the machine code of the LOAD DSS instruction. Thus, for LOAD DSS the update of the signature S is: $S_{i+1} = S_i\alpha \oplus I_i \oplus DSS_i$, where $S\alpha$ is the current signature shifted by the MISR, I is the LOAD DSS instruction's machine code, *DSS* is the data-structure signature that is read from memory, and $\oplus$ is GF(2) addition (XOR). For other instructions, the signature is updated in the usual manner: $S_{i+1} = S_i\alpha \oplus I_i$.

LOAD DSS can significantly lower performance overhead and memory overhead for DSS monitoring. Compared with the software-only checking sequence shown above, LOAD DSS allows the reference signature LOAD, the COMPARE, and the BRANCH to be eliminated, because the corresponding functionality is provided when the signature is checked at the next reference signature. Thus, DSS monitoring is reduced to one extra instruction to check each data access, as shown below. Checking data accesses in this manner will lower performance overhead to an estimated 25% for the data in [4].

```
Load_DSS R0 <- [p-1]            /* signature the
                                   DSS */
Load (Store) Rz <- (->) [p]     /* do usual
                                   access */
```

The second architecture support technique exploits two natural redundancies to further lower the performance overhead for monitoring data-structure signatures. To simplify the presentation of this technique, we assume that the processor has separate first-level instruction and data caches. First, the data-access mechanism is often idle, during roughly 75% of the cycles for the processors studied in [4]. A time redundancy technique could potentially use these idle cycles for data-access checking. Second, the instruction set architecture includes a natural redundancy in the data-access instructions. For an architecture with a w-bit address, more than w bits specify a load or store address. For displacement addressing mode, the architecture specifies an address using w bits in the base register and k bits in the instruction's offset field. Similarly, indexed addressing mode uses w bits from the base register and w bits from the index register to specify an address. Each data-access instruction effectively specifies two addresses: the address in the base register, and the computed address produced by adding the base register to the offset field or the index register. The computed address points to the target data, and the naturally redundant base address points to the data structure's first word.

The architecture is extended to include two new instructions, *checked load* and *checked store*. A checked load/store performs two memory operations: It loads/stores from/to the computed address, and it reads the DSS at an offset of –1 from the naturally redundant base-register address. For a checked load/store the update of the signature S is: $S_{i+1} = S_i\alpha \oplus I_i \oplus DSS_i$. Thus, checked load/stores allow data structure signatures to be monitored without separate checking instructions, and instruction memory overhead for DSS monitoring can be eliminated.

Checked load/stores also allow a significant reduction in performance overhead. A checked load/store occupies the data-access mechanism for two cycles, one cycle to load/store the data and one cycle to read the DSS. If the instruction following a checked load/store is a non-data-access instruction, the checked load/store can complete its

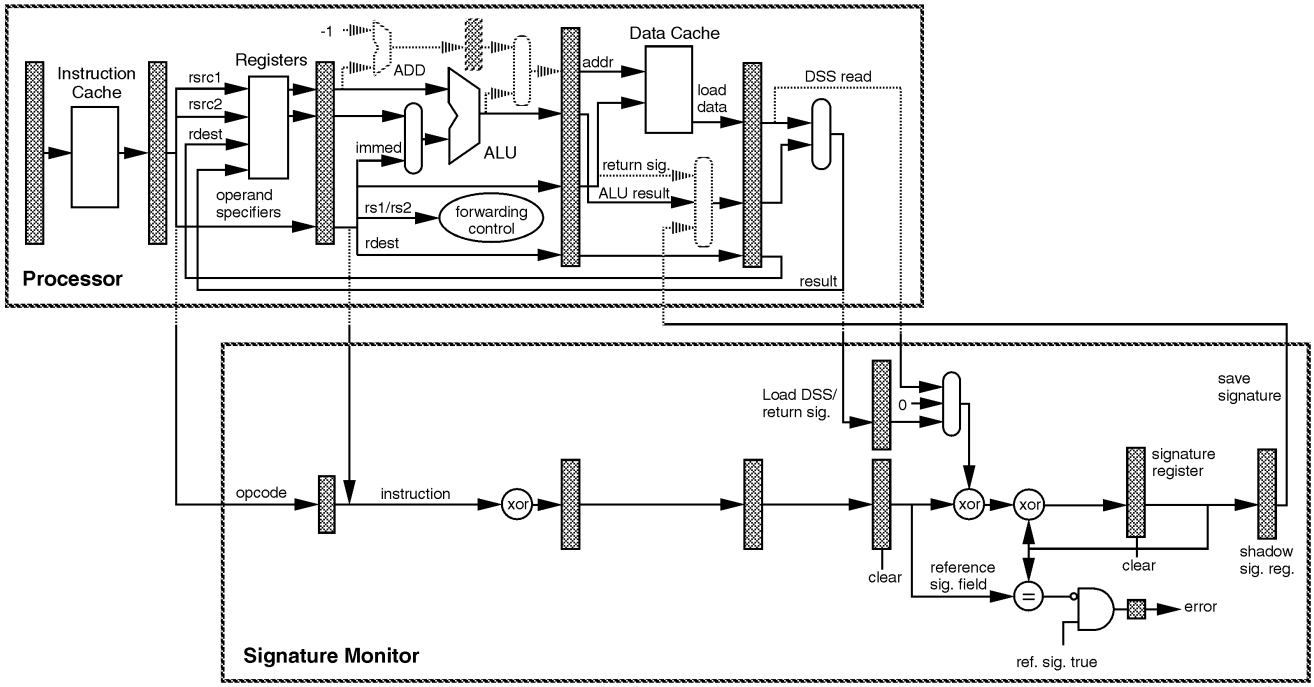


Fig. 4. Processor and monito with support for load DSS and checked load/store.

second data access while the non-data-access instruction occupies the data-access pipeline stage. If the instruction following a checked load / store is a data-access instruction, the pipeline must stall to allow the DSS read to complete. An initial estimate of the performance overhead due to checked load / store pipeline stalls can be made by assuming that data accesses are uniformly distributed. The instruction mix in [4] includes 25% data-access instructions. Thus, the estimated performance overhead is $0.25 * 0.25 = 6\%$. This compares with an estimated 25% performance overhead using LOAD DSS, and about 75% overhead without architecture support. Section 4 shows how the compiler can schedule LOAD DSS and checked load / store instructions in combination to further reduce performance overhead for DSS monitoring.

3.3 Implementation of Load DSS and Checked Load/Store

Fig. 4 illustrates an implementation of a monitor and a processor that includes support for LOAD DSS and checked load / stores. The figure uses dotted lines to show the components and the interconnect within the processor that support DSS monitoring. The monitor's signature register and the shadow signature register are placed at the end of the pipeline and latches are added to the intervening monitor pipeline stages. This allows the data-structure signatures, which are accessed from the data cache, to be added to the signature during the same cycle that the corresponding LOAD DSS or checked load / store is added to the signature. A set of XOR gates is added toward the beginning of the monitor's pipeline that partially compresses the instruction, without obscuring the reference signature field. A decrementer and a latch are added to the processor's ALU pipeline stage. The decrementer produces the DSS address for a checked load / store instruction, and the latch delays the DSS address one cycle. This delay allows the DSS read-operation produced by a checked load / store instruction to occur in the cycle after the instruction's load / store operation. A multiplexer is added in the ALU pipeline stage to allow the next data-access address to be the usual value from the ALU, or the delayed DSS address of a checked load / store. Two latches in the monitor delay the partially compressed instruction and the LOAD DSS result relative to the DSS read-result. Hence, the DSS read-result can be added to the signature during the same cycle as the machine code of the corresponding checked load / store. A three-input multiplexer is used in the monitor to allow the monitor to selectively add the LOAD DSS result or the DSS read-result to the signature.

The implementation uses a three-input multiplexer in the data-access stage of the processor's pipeline to facilitate saving and restoring the signature around a context switch. When the exception handler executes MOVSIG, the signature from the shadow signature register is routed through this multiplexer along the path to the register file. When MRFE is executed, the return signature is routed through this multiplexer along the path to the first set of XOR gates in the MISR stage of the monitor's pipeline. When the MRFE reaches the MISR stage, the signature register is cleared, and the monitor latch that inputs the partially compressed instruction to the MISR stage is also clear. Thus, the value that is placed into the signature register is the return signature.

The hardware support for LOAD DSS and checked load / store adds an additional 2,758 transistors to the monitor and the processor. The total of 4,062 transistors that are needed for monitoring instructions and data-structure signatures is 1% or less of the complexity of a typical RISC processor.

3.4 Software Faults Detected

DSS monitoring can detect faults caused by several types of programming defects, including:

- **Allocation Management**: The program deallocates a structure while active pointers to the structure still exist.
- **Copy Overrun**: The program copies past the end of an array.
- **Page-Table Management**: An entry in the virtual-memory page table points to the wrong physical page.
- **Pointer Management**: A variable containing a data address (a pointer) is damaged.
- **Register Reuse**: In assembly language code, a register is reused without saving and restoring a live pointer that was in the register.
- **Type Mismatch**: A field is added to or deleted from a structure but not all code that uses the structure is updated to reflect the change.
- **Uninitialized Pointer**: A pointer is used before it is initialized.

Data-access faults due to page-table management, pointer-management, register reuse, and uninitialized pointer defects are detectable using DSS monitoring as described thus far. Data-access faults caused by allocation management defects can be detected by clearing the DSS when the data structure is deallocated. To detect data-access faults caused by type mismatch, the compiler assigns a new DSS to a data structure when a programmer redefines the data structure. Thus, code that is not updated will have the old, incorrect DSS. Copy overruns are detectable because the first corrupted word is the DSS, and the check of the DSS before the next access of the damaged data structure will detect the overrun.

The proposed DSS monitoring approach also allows array accesses to be checked with low performance overhead. For an array that is accessed using indexed addressing mode, faults in the pointer to the base of an array are detectable as described above. However, access errors due to faults in the index, i.e., bound violations, are not detected. Detecting bound violations can require a sequence of several instructions: COMPARE index, array-min; BRANCH LESS THAN; COMPARE index, array-max; BRANCH GREATER THAN. Bounds-checking performance overhead is often unacceptable. DSS monitoring can provide low-performance-overhead detection of index faults by transforming an array of simple elements, e.g., integers, into an array of structures, with each element containing a DSS. At each array access, the DSS is checked in the proposed manner. The tradeoff is low-performance-overhead array access checking for increased data space for the array.

Sullivan and Chillarege [31] report statistics on overlay faults, the subset of software data-access faults that causes errant writes to memory. They studied software defects in the IBM MVS operating system reported between 1986 and 1989. They extracted a sample of 150 defects that caused system failures from the IBM field service database, and manually classified the defects by type based on the description of the design change that was made to correct the defect. As Table 1 shows, overlay faults caused 42% of all system failures.

TABLE 1
FROM [31]: TYPES OF SOFTWARE DEFECTS THAT CAUSE OVERLAY FAULTS, AND THE PERCENTAGE OF SYSTEM FAILURES THE DEFECT TYPE PRODUCES

Defect Type	System Failures
Allocation Management	7%
Copy Overrun	2%
Page-Table Management	12%
Pointer Management	9%
Register Reused	3%
Type Mismatch	1%
Uninitialized Pointer	8%
Total	42%

3.5 Hardware Faults Detected

DSS monitoring allows detection of hardware data-access faults at all levels of the memory hierarchy, faults that are not detected by conventional signature monitoring. Example hardware faults that can be detected are described below.

Register Faults. All transient register faults are detectable that involve a register that contains a data address, including:

1) a fault in one of the bits of a register,
2) a register-access fault that causes an incorrect register to be read,
3) a register-access fault that causes a data address to be written to the wrong register, and
4) a register-access fault that causes a data address to be overwritten.

Note that a register-access fault can occur within the register file (e.g., within the select logic), or can occur in one of the bits in a pipeline latch that passes the destination register specifier down the pipeline (see Fig. 4). Transient register faults that involve an instruction address (e.g., a subroutine return address) are detectable by conventional signature monitoring. Because data addresses appear in the register file much more often than instruction addresses, coverage of faults in the register file increases significantly.

Pipeline Faults. All transient pipeline faults that damage the base-register address of a load or a store can be detected, including:

1) a fault in one of the bits in the operand pipeline latch that passes the base-register address from the register-access stage to the execution stage (see Fig. 4),
2) a fault in the forwarding datapath or the forwarding control logic (see [12]) when the base-register address is forwarded within the pipeline from a previous instruction,
3) a fault in the result pipeline latch or the result multiplexer that damages a pointer fetched from memory as it is passed along to the register file,
4) a fault in the ALU that produces an incorrect address when a pointer is loaded from memory, or a fault in the address latch when a pointer is loaded from memory.

These latter faults will be detected when a data access is made using the damaged pointer.

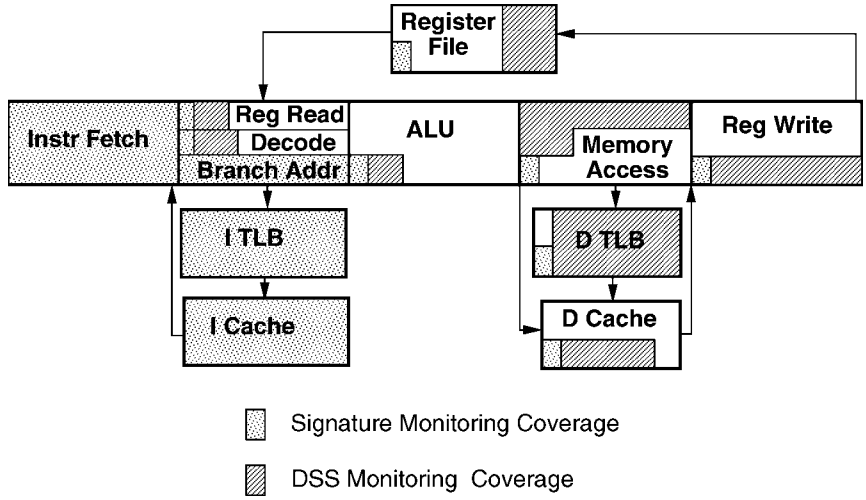


Fig. 5. Areas where faults are covered by signature monitoring and DSS monitoring.

Translation Look-Aside Buffer (TLB) Faults. A TLB contains an array of virtual page numbers and a corresponding array of physical page numbers that cache the most recent virtual to physical page translations [12]. DSS monitoring can detect most of the transient faults that occur in the virtual or physical page numbers in the data-cache TLB. (Conventional signature monitoring can detect such faults in the instruction-cache TLB.) Due to locality, the DSS and the target data are very likely to be on the same page in virtual memory. Thus, a TLB virtual or physical page-number fault that causes a data-access error for the target data is also very likely to be manifested when the DSS is read, and the fault will be detected. Similarly, DSS monitoring can detect a multiple-cycle fault in the TLB control logic if the duration of the fault spans the data access and the DSS read, and the fault produces an incorrect physical page number for both accesses. A transient fault that occurs in the *page translation table* [12] that is stored in main memory is also detectable. DSS monitoring provides an attractive alternative to the TLB fault detection technique proposed by Chang and Saxena [3], which requires error detection bits to be attached to each virtual and physical page number in the TLB, and requires a dedicated error detection mechanism in the TLB.

Memory System Faults. DSS monitoring also allows detection of certain faults that occur in the data cache and in main memory. A fault in a data address that is stored in the data cache or in main memory can be detected. DSS monitoring can detect a multiple-cycle fault in the data cache that spans the data access and the DSS read, and causes each to access an incorrect cache block. A transient fault that causes an incorrect block to be transferred between main memory and the data cache will likely be detected because, due to locality, the target data and the data structure signature are both likely to be on the faulty cache line. Similarly, a transient fault in one of the cache tag bits that causes an erroneous cache hit is likely to be detected.

Fig. 5 is an illustration of the coverage provided in each processor module by signature monitoring and by DSS monitoring. The size of the shaded region in each module is in proportion to the projected coverage for that module. The signature-monitoring approach targets instruction-access faults and instruction-memory faults. Signature monitoring provides nearly complete coverage for faults that occur in the instruction cache, the instruction-cache TLB, and the dedicated branch-address adder. Limited coverage is provided for the remaining modules because only a fraction of the faults in these modules are instruction-access faults. Instruction-access faults can occur when these modules handle an instruction address that is used for a register indirect jump or for a subroutine return, but this is relatively infrequent. Fault injection experiments on RISC processors (not including an instruction cache or Icache TLB) suggest that 30-60% of all faults are instruction-access faults [6], [22], [36]. The DSS monitoring approach targets data-access faults, which are not detected by signature monitoring. DSS monitoring provides a significant increase in fault coverage for the ALU, the data-access mechanism (including the Dcache TLB and the data cache), and the register file, because these modules handle data addresses much more often than instruction addresses. The faults that cannot be detected are primarily pure data faults, which are not targeted by either approach.

4 Scheduling DSS Monitoring Instructions

This section presents an instruction scheduling algorithm that allows the compiler to significantly reduce the performance overhead for DSS monitoring by simultaneously scheduling checked load / store and LOAD DSS instructions.

Instruction scheduling is a compiler optimization that attempts to avoid pipeline stalls by rearranging the code sequence. A pipeline stall occurs for a *data hazard* [12], when an instruction needs to use a value that is produced by a preceding instruction still in the pipeline, and the value is not yet available. Similarly, a pipeline stall occurs for a *structural hazard* [12], when an instruction needs to use a hardware unit, but the unit is busy.

Instruction scheduling can either be *local*, within a basic block, or *global*, beyond the basic block [1]. The focus here is on local instruction scheduling, the method typically used in practice. Compiler optimizations interact, especially

instruction scheduling and register allocation, and various orderings of optimization phases are possible. Scheduling DSS monitoring instructions after register allocation improves data-access checking because register spills and restores to and from memory can be checked. Thus, the following discussion assumes DSS monitoring instructions are scheduled after register allocation.

4.1 Checked Load/Store and Load DSS Instruction Scheduling

The compiler can minimize DSS-monitoring performance overhead by scheduling a combination of checked load/store and LOAD DSS instructions. A checked load/store can be scheduled at zero performance overhead and zero memory overhead if a non-data-access instruction can be scheduled after the checked load/store. If not, scheduling a checked load/store would cause performance overhead due to the pipeline stall. This overhead can be avoided by using a LOAD DSS to check the load or store (as shown in Section 3.2), if the LOAD DSS can be scheduled at an otherwise idle cycle before or after the load or store. A LOAD DSS can exploit idle cycles because it is more mobile than a checked load/store. A LOAD DSS only has data dependencies for its base register, while a checked load/store has data dependencies for its base register and for its source and destination registers.[1] Also, unlike LOAD DSS, a checked load/store may potentially *alias* [12] another data access, preventing the compiler from moving the checked load/store past the data access. Aliasing occurs when there are multiple ways to refer to the address of a variable.

An extension of the *list scheduling* [5] is proposed that efficiently schedules checked load/store and LOAD DSS instructions. List scheduling is the traditional approach to local instruction scheduling that operates on:

1) a directed acyclic graph (DAG), where nodes in the graph represent instructions, arcs in the graph represent data dependencies between instructions, and an arc's weight represents the latency the pipeline requires between the corresponding instructions to avoid a data hazard; and
2) a list of nodes ordered by priority.

A node is *ready* if all its predecessors have been scheduled, the predecessors' latency requirements are satisfied, and no structural hazards exist. Each scheduling iteration schedules a ready node with the highest priority, or schedules an idle cycle if no node is ready. Various priority assignment heuristics have been proposed [18]. For RISC processors, an optimal priority assignment is produced using the polynomial-time *rank algorithm* [23].

The proposed scheduling algorithm uses list scheduling to initially schedule a checked load/store to check each data access. However, when two checked load/stores are scheduled consecutively, which would cause a pipeline stall, an attempt is made to schedule a LOAD DSS that checks the first load/store at an idle cycle that occurs before the load/store. If no idle cycle is found, list scheduling continues, and an attempt is made to schedule a LOAD DSS at an idle cycle that occurs after the load/store. If a LOAD DSS is scheduled at an idle cycle in either case, a normal load/store instruction replaces the scheduled checked load/store. If no idle cycle is found at the scheduling iteration where the LOAD DSS base register is redefined or where the basic block ends, the scheduled checked load/store remains. The detailed scheduling algorithm is given below, with bold step numbers indicating the traditional list scheduling steps.

Scheduling Algorithm

1 Change each load/store that requires checking to a checked load/store

2 Prioritize nodes and sort nodes into a list with nondecreasing priority

3 Create a sorted, prioritized, auxiliary list that contains a LOAD DSS_i for each checked load/store *i*, with priority equal to that of the highest priority instruction that redefines the base register of LOAD DSS_i, or equal to 0 if the base register is not redefined. Mark each LOAD DSS_i not ready.
4 TIME_SLOT = 0
5 While the node list is not empty
6 TIME_SLOT = TIME_SLOT + 1
7 **If** a node is ready **then**
8 Remove the first ready node *R* and schedule *R* at TIME_SLOT
9 Remove from the auxiliary list any ready LOAD DSS whose base register is redefined by *R*
10 **Else If** (the last scheduled node *L* is a checked load/store) and ((the predecessors of a data access node *j* are all scheduled) and (the latency requirements of all predecessors to node *j* are satisfied)) **then**
11 Remove the first such data access node *j* and schedule at TIME_SLOT
12 Remove from the auxiliary list any ready LOAD DSS whose base register is redefined by *j*
13 IDLE_CYCLE = FIND_FIRST_IDLE_CYCLE(*L*)
14 **If** IDLE_CYCLE != nil **then**
15 Schedule LOAD DSS_L at IDLE_CYCLE
16 Remove LOAD DSS_L from auxiliary list
17 Change node *L* to a load/store
18 **Else** mark LOAD DSS_L ready
19 **Else if** a LOAD DSS is ready **then**
20 Remove first ready LOAD DSS_i from auxiliary list and schedule at TIME_SLOT
21 Change node *i* to a load/store
22 **Else** Schedule an idle cycle at TIME_SLOT

The function FIND_FIRST_IDLE_CYCLE identifies the earliest idle cycle in the instruction schedule (if any) that exists between the node that defines the base register of the given checked load/store node and the checked load/store node. The next subsection describes an efficient implementation of FIND_FIRST_IDLE_CYCLE.

For illustration, the new scheduling algorithm is applied to the example data dependency graph in Fig. 6, where instructions have the form `operation destination, source1, source2`; and where `X(rY)` is the effective memory address (the contents of base register `rY` plus the offset `X`). Each arc in the figure is labeled with the required latency between the corresponding instructions. Each node is labeled with a priority based on the node's *critical path*, the longest path from the node to any leaf node in the

1. An instruction B is data dependent on another instruction A if A defines one of B's operands and/or B defines one of A's operands, and A appears before B in the original instruction sequence. The instruction scheduler cannot change the relative order of dependent instructions.

graph. For this example we assume there is no aliasing and that each load and store is checked. The algorithm starts by following traditional list scheduling for the first five iterations, scheduling MULT, NOP, ADD, ADD, CHECKED LOAD in the first five time slots. During the sixth iteration, either one of the CHECKED STOREs is scheduled in time slot 6 (they have the same priority and both are ready). Because the CHECKED STORE would cause the CHECKED LOAD in time slot 5 to stall, the CHECKED LOAD is converted to a LOAD. A LOAD DSS is created and an attempt is made to schedule the LOAD DSS at an idle cycle before the LOAD. The attempt fails because there is no idle cycle between the ADD in time slot 4 that defines the base register of the LOAD and the LOAD in time slot 5. So, the LOAD DSS is marked ready. During the seventh iteration, the second CHECKED STORE is scheduled and the scheduled CHECKED STORE in time slot 6 is converted to a STORE. A LOAD DSS is created and an attempt is made to schedule the LOAD DSS at an idle cycle before the STORE. The LOAD DSS can be scheduled at time slot 2, because the base register of the STORE is defined in a previous basic block. At iteration nine, there is no ready program instruction, so the ready LOAD DSS that corresponds with the LOAD in time slot 5 can be scheduled in time slot 9. The resulting schedule is shown in Fig. 7. DSS monitoring causes no performance overhead for this example: One data access is checked by a LOAD DSS scheduled at an otherwise idle cycle before the data access, one data access is checked by a LOAD DSS scheduled at an otherwise idle cycle after the data access, and one data access is checked using a checked load/store that is not followed by a data access.

4.2 Complexity of the Scheduling Algorithm

The proposed extension to list scheduling has the same algorithm complexity (worst case running time) as basic list scheduling. The minimum complexity for basic list scheduling is set by the sorting operation, $O(n \log n)$, although the priority assignment algorithm can increase the algorithm complexity, e.g., [23]. The proposed scheduling algorithm includes an additional $O(n \log n)$ sorting operation. Otherwise each new scheduling step is $O(n)$, with the possible exception of FIND_FIRST_IDLE_CYCLE. Implementing FIND_FIRST_IDLE_CYCLE in a straightforward manner results in an algorithm complexity of $O(n^2)$ for scheduling DSS monitoring instructions, higher than that of basic list scheduling. Thus, a more efficient implementation of FIND_FIRST_IDLE_CYCLE is useful.

An efficient algorithm for FIND_FIRST_IDLE_CYCLE is proposed that has $O(n)$ complexity. The algorithm uses an integer array `Reg[]` that contains one element for each processor register. During scheduling, when an instruction is scheduled that posts a result to register i, `Time_Slot` is written to `Reg[i]`. When an idle cycle is needed for a LOAD DSS_L associated with a checked load/store L using base register k, a linear search begins at the time slot that is stored in `Reg[k]` plus one. The search ends at the first idle cycle where a LOAD DSS can be scheduled, with the corresponding time-slot number produced as the result. Otherwise, the search ends at node L and a `nil` result is produced. In each case, `Reg[k]` is updated with the time slot where the search ends. The algorithm is $O(n)$ because for each `Reg[i]` the total number of time slots searched is at most n, because `Reg[i]` is written at most n times, and because the number of elements in `Reg[]` is a constant.

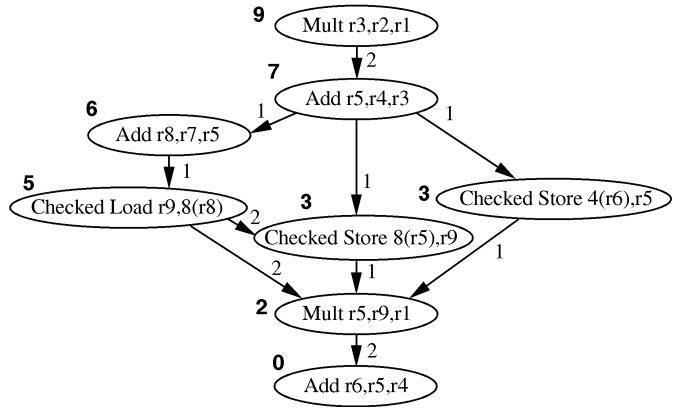


Fig. 6. Example data dependency graph.

1.	Multiply	r3, r2, r1
2.	Load DSS	r0, 0(r6)
3.	Add	r5, r4, r3
4.	Add	r8, r7, r5
5.	Load	r9, 8(r8)
6.	Store	4(r6), r5
7.	Checked Store	8(r5), r9
8.	Multiply	r5, r9, r1
9.	Load DSS	r0, 0(r8)
10.	Add	r6, r5, r4

Fig. 7. Instruction schedule for example dependency graph.

5 Experimental Results

This section presents experimental results for an emulation of DSS monitoring. The results show that DSS monitoring can achieve high data-access checking coverage, while causing little performance overhead.

5.1 Experimental Method

An experimental system was built that emulates DSS monitoring extensions for the MIPS R3000 architecture [15]. A LOAD DSS is emulated using a normal R3000 LOAD, with the result written to $\mathtt{r}_0$ which is hardwired to the value zero. A CHECKED LOAD is emulated by a normal LOAD and a CHECKED STORE is emulated by a normal STORE. A NOP is placed where a checked load/store is followed by another data-access instruction, to emulate the pipeline stall caused by the structural hazard.

Version 2.3 of the GNU C Compiler (GCC) [30] was modified to allow an arbitrary `C` language program to be compiled into DSS-monitoring machine code for the extended architecture. Modifications include changes to GCC's frontend that expand the size of each data structure to include space for the data structure signature. Changes to GCC's backend include modifications to the instruction scheduler that implement checked load/store and LOAD DSS scheduling. Backend changes also include expanding the size of each stack frame to include a DSS. Stack-frame DSSs allow accesses through the stack pointer to be checked.

The code produced by the modified GCC compiler was executed on the MIPS-R3000 based DECStation 5000/200.

TABLE 2
DESCRIPTION OF THE SPEC92 INTEGER BENCHMARK APPLICATIONS

Program	Description
compress	Adaptive Lempel-Ziv compression
eqntott	Truth table builds from Boolean expressions
espresso	Boolean function minimization
sc	Spreadsheet calculator
xlisp	Lisp interpreter solving the nine-queens problem

TABLE 3
DSS MONITORING RESULTS FOR STRUCTURE ACCESSES

SPEC92 Integer Benchmark	Access Checking Coverage	Performance Overhead		Memory Overhead	
		Checked L/S and Load DSS	Existing Instructions	Checked L/S and Load DSS	Existing Instructions
compress	65.4%	2.5%	78.5%	0.0%	0.6%
eqntott	88.2%	0.4%	49.9%	0.0%	0.7%
espresso	81.4%	1.6%	74.0%	31.2%	38.1%
sc	81.2%	1.8%	28.5%	0.9%	4.6%
xlisp	96.8%	10.9%	146.1%	15.6%	52.2%
Mean	82.6%	3.4%	75.4%	9.6%	19.2%

The commercial profiling tool **pixie** [14] was used to gather dynamic instruction-execution counts for a compiled version of a program with DSS monitoring and for a version without DSS monitoring, to determine performance overhead for DSS monitoring. Custom instrumentation was created to measure the dynamic count of data-access checks and the count of total data accesses, to determine checking coverage. Custom instrumentation was also created to measure memory overhead for statically and dynamically allocated data structures. Dynamically allocated data was measured by modifying the memory allocator (**malloc**) to track the maximum size of the heap.

GCC was also modified to emulate DSS monitoring without architecture support, so that the corresponding performance overhead could be estimated. A sequence of four dummy instructions was placed prior to each memory access to emulate the checking sequence described in Section 3.1. GCC was also modified to globally reserve two registers for use in the DSS/reference-signature comparison. Loop-invariant code motion of the reference-signature LOADs was not implemented, thus the performance overhead results are conservative.

DSS monitoring was evaluated using programs from the industry-standard SPEC92 integer benchmark suite [8]. The five programs shown in Table 2 were used in the evaluation. **compress**, **eqntott**, and **xlisp** compiled and ran successfully with data-structure signatures inserted. **espresso** and **sc** initialize arrays of structures in a manner that depends on programmer assumptions about the size of the structures. Such assumptions are not compatible with the compiler generated data-structure signatures, because the size of the data structure increases. However, such assumptions also violate the ANSI C standard, which allows an implementation-dependent number of padding bytes to be placed between elements in a structure [17]. A simple change to the initialization source code corrected this incompatibility for **espresso** and for **sc**. The sixth SPEC92 integer benchmark, **gcc**, did not run with data-structure signatures inserted because it relies heavily on assumptions about the size of structures. Thus, **gcc** was not used as a benchmark in this study.

5.2 Results

Experiments were performed to evaluate DSS monitoring for each of three combinations of high-level-language data access methods:

1) structure accesses, e.g., **p.element_one**,
2) structure accesses plus array accesses, and
3) all data accesses: structure accesses, array accesses, and pointer dereferences (e.g., ***p**).

Array accesses are broken out separately to show the trade-off between access-checking coverage and memory overhead, when arrays are checked as described in Section 3.4. For simple data types (e.g., char, int, float) that are accessed using a pointer dereference (e.g., ***p**), the C programming model for DSS monitoring would require the programmer to declare the variable as a single-element structure for a DSS to be inserted. Pointer dereferences are broken out separately because such declarations are not made for these benchmarks, and thus memory overhead for pointer-dereference accesses cannot be accurately measured.

Table 3 shows the results for DSS monitoring applied to structure accesses. Access checking coverage is high, at 83%. Access checking coverage is the fraction of checked data accesses, relative to all data accesses. Performance overhead using the proposed architecture support is very low, at 3%, while the performance overhead using the existing instructions is relatively high, at 75%. Performance overhead is the relative increase in dynamic instruction count. On a real system, data-cache misses and instruction-cache misses will cause additional performance overhead that depends on the cache's size and organization. The performance overhead due to data-cache misses is likely to be similar with and without architecture support because they use an equal number of data accesses for DSS monitoring. The performance overhead due to instruction-cache misses will be much lower with architecture support because there are significantly fewer DSS monitoring instructions.

Memory overhead is low using architecture support, at 10%, and is moderate without architecture support, at 19%. Memory overhead is the incremental increase in the total size of memory used with DSS monitoring (including code, the run-time stack, the heap, and the global data region)

TABLE 4
DSS MONITORING RESULTS FOR STRUCTURE ACCESSES AND ARRAY ACCESSES

SPEC92 Integer Benchmark	Access Checking Coverage	Performance Overhead		Memory Overhead	
		Checked L/S and Load DSS	Existing Instructions	Checked L/S and Load DSS	Existing Instructions
compress	86.1%	7.0%	93.1%	119.3%	119.9%
eqntott	88.2%	0.4%	49.9%	9.9%	10.6%
espresso	83.9%	3.2%	76.0%	32.2%	39.1%
sc	94.8%	1.7%	30.5%	14.8%	18.6%
xlisp	96.8%	13.1%	146.3%	16.9%	53.5%
Mean	90.0%	5.1%	79.2%	38.6%	48.3%

TABLE 5
DSS MONITORING RESULTS FOR ALL ACCESSES

SPEC92 Integer Benchmark	Access Checking Coverage	Performance Overhead	
		Checked L/S and Load DSS	Existing Instructions
compress	100%	8.7%	114.9%
eqntott	100%	1.1%	57.7%
espresso	100%	3.3%	90.4%
sc	100%	1.7%	32.4%
xlisp	100%	13.4%	152.0%
Mean	100%	5.6%	89.4%

relative to the total size of memory used without DSS monitoring.

Table 4 shows coverage, performance overhead, and memory overhead results for DSS monitoring applied to structure access and array accesses. Monitoring array accesses increases access-checking coverage by 7%, while causing performance overhead to increase by only 2% using architecture support. Memory overhead increases significantly because one DSS is added per array element. Note that overhead can exceed 100% when the size of an array element is less than the four-byte DSS. For example, `compress` includes arrays with one-byte character (`char`) elements.

Table 5 shows the performance overhead results for DSS monitoring applied to all accesses: structure accesses, array accesses, and pointer dereferences. Memory overhead results are not meaningful in this case because a DSS is not allocated for simple data types that are accessed through pointer dereferences. However the performance overhead for checking pointer dereferences can be emulated. The table shows that using checked load/store and Load DSS architecture support, DSS monitoring can be applied to all data accesses while causing only 6% performance overhead. This compares with hundreds of percent overhead for the prior data-access checking techniques that do not use architecture support [2], [11]. Even without architecture support, the 90% performance overhead for DSS monitoring is significantly lower than the prior software-only techniques [2], [11].

6 SUMMARY

This paper proposes a new approach that allows signatures stored in data memory to be monitored along with instruction signatures, providing low-cost concurrent detection of software and hardware data-access faults. Low-cost concurrent detection is achieved by exploiting four separate natural redundancies that exist in the program and in the computer system: spatial locality in data structures, time redundancy in the data access mechanism, address specification redundancy in the architecture, and redundancy in program control flow. A new redundant data structure is proposed that includes a data-structure signature that allows accesses to the data structure to be concurrently monitored for software and hardware data access faults. Simple architecture support in the form of LOAD DSS and checked load/store instructions allows data structure signatures to be monitored with low performance overhead. The architecture support causes an increase of only 2,800 transistors for an efficient CMOS implementation of a signature monitor. Performance overhead is further reduced using an instruction scheduling algorithm that extends traditional list scheduling to include LOAD DSS and checked load/store scheduling. For the SPEC92 integer benchmarks, all structure accesses (accounting for 83% of all data accesses) can be checked while causing an average performance overhead of only 3%, and an average memory overhead of only 10%. DSS monitoring can check all data accesses while causing only 6% performance overhead. Unlike the 100s of percent performance overhead caused by prior software-only approaches, the level of performance overhead caused by DSS monitoring is suitable for concurrent detection of data-access faults.

The proposed DSS monitoring approach complements and can be used in combination with other existing low-cost concurrent detection approaches. The proposed approach does not target pure data faults that can occur in the ALU. Such faults can be detected using the ALU time-redundancy approach proposed in [28], which exploits unused ALU cycles. The proposed DSS monitoring approach can detect damaged links (pointers) within data structures such as singly linked lists, unlike the robust data structure approach proposed by Taylor, Morgan, and Black [32]. Also, DSS monitoring provides low-cost concurrent detection of damaged links and provides fault location information, which are not provided by the robust data structure approach. However, unlike the robust data structure approach, DSS monitoring can detect but cannot correct damaged links.

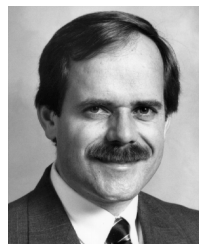
Using DSS monitoring in combination with robust data structures allows low-cost concurrent detection and correction of data-structure links that are damaged by software faults.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation under grant CCR-93-12880 and is based on "Efficient Memory-Access Checking" by Kent D. Wilken and Timothy Kong which appeared in the Proceedings of the International Symposium on Fault-Tolerant Computing, pp. 566-575, 1993.

REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.
[2] T. Austin, S. Breach, and G. Sohi, "Efficient Detection of All Pointer and Array Access Errors," *Proc. Conf. Programming Language Design and Implementation*, pp. 290-301, 1994.
[3] D. Chang and N. Saxena, "Concurrent Error Detection/Correction in the HaL MMU Chip," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 630-635, 1993.
[4] R. Cmelik, S. Kong, D. Ditzel, and E. Kelly, "An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 290-302, 1991.
[5] E. Coffman, *Computer and Job-Shop Scheduling Theory*. Wiley, 1976.
[6] E. Czeck and D. Siewiorek, "Effects of Transient Gate-Level Faults on Program Behavior," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 236-243, 1990.
[7] X. Delord and G. Saucier, "Formalizing Signature Analysis for Control Flow Checking of Pipelined RISC Microprocessors," *Proc. Int'l Test Conf.*, pp. 936-945, 1991.
[8] K. Dixit, "New CPU Benchmark Suites from SPEC," *Digest of Papers Compcon, Spring 1992*, pp. 305-310, 1992.
[9] D. Dobberphul et al., "A 200-mhz 64-b Dual-Issue CMOS Microprocessor," *IEEE J. Solid State Circuits*, vol. 27, no. 11, pp. 1,555-1,567, Nov. 1992.
[10] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-Ion Radiation," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 340-347, 1989.
[11] R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," *Proc. Winter Usenix Conf.*, pp. 125-136, 1992.
[12] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, second edition. Morgan Kaufmann, 1996.
[13] IBM, *IBM RISC System/6000 Technology*. IBM, 1990.
[14] MIPS Computer Systems Inc., *Language Programmer's Guide*, 1986.
[15] G. Kane, *MIPS RISC Architecture*. Prentice Hall, 1988.
[16] W. Kao, R. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults," *IEEE Trans. Software Eng.*, vol. 19, no. 11, pp. 1,105-1,118, Nov. 1993.
[17] B. Kernighan and D. Ritchie, *The C Programming Language*. Prentice Hall, 1988.
[18] S. Krishnamurth, "A Brief Survey of Papers on Scheduling for Pipelined Processors," *SIGPLAN Notices*, vol. 25, no. 7, pp. 97-106, July 1990.
[19] A. Mahmood and E. McCluskey, "Concurrent Error Detection Using Watchdog Processors—A Survey," *IEEE Trans. Computers*, vol. 37, no. 2, pp. 160-174, Feb. 1988.
[20] B. Miller, L. Fredriksen, and B. So, "An Empirical Study of UNIX Utilities," *Comm. ACM*, vol. 33, no. 12, pp. 32-44, Dec. 1989.
[21] M. Namjoo, "Techniques for Concurrent Testing of VLSI Processor Operation," *Proc. Int'l Test Conf.*, pp. 461-468, 1982.
[22] J. Ohlsson, M. Rimen, and U. Genneflo, "A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-In Watchdog," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 316-325, 1992.
[23] K. Palem and B. Simons, "Scheduling Time-Critical Instructions on RISC Machines," *ACM Trans. Programming Languages and Systems*, vol. 15, no. 4, pp. 632-658, Sept. 1993.
[24] N. Saxena and E. McCluskey, "Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums," *IEEE Trans. Computers*, vol. 39, no. 4, pp. 554-558, Apr. 1990.
[25] M. Schuette and J. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Trans. Computers*, vol. 36, no. 3, pp. 264-276, Mar. 1987.
[26] D. Siewiorek and R. Swarz, *Reliable Computer Systems: Design and Evaluation*. Digital Press, 1992.
[27] R. Sites, *Alpha Architecture Reference Manual*. Digital Press, 1992.
[28] G. Sohi, M. Franklin, and K. Saluja, "A Study of Time-Redundant Fault Tolerant Techniques for High-Performance Pipelined Computers," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 436-443, 1989.
[29] T. Sridhar and S. Thatte, "Concurrent Checking of Program Flow in VLSI Processors," *Proc. Int'l Test Conf.*, pp. 191-199, 1982.
[30] R. Stallman, *Using and Porting GNU CC*. Free Software Foundation, Inc., 1992.
[31] M. Sullivan and R. Chillarege, "Software Defects and Their Impact on System Availability—A Study of Field Failures in Operating Systems," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 2-9, 1991.
[32] D. Taylor, D. Morgan, and J. Black, "Redundancy in Data Structures: Improving Software Fault Tolerance," *IEEE Trans. Software Eng.*, vol. 6, no. 6, pp. 585-594, Nov. 1980.
[33] J. Wang, S. Fang, and W. Feng, "New Efficient Designs for XOR and XNOR Functions on the Transistor Level," *IEEE J. Solid State Circuits*, vol. 29, no. 7, pp. 780-786, July 1994.
[34] N. Warter and W. Hwu, "A Software Based Approach to Achieving Optimal Performance for Signature Control Flow Checking," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 442-449, 1990.
[35] K. Wilken, "An Optimal Graph-Construction Approach to Placing Program Signatures for Signature Monitoring," *IEEE Trans. Computers*, vol. 42, no. 11, pp. 1,372-1,381, Nov. 1993.
[36] K. Wilken, A. Barr, and R. Hoskin, "A RISC Architecture for Concurrent Error Detection," Technical Report ECE-CERL-TR-95-06-01, Univ. of California at Davis, June 1995.
[37] K. Wilken and J. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent-Detection of Processor Control Errors," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 3, pp. 629-641, June 1990.
[38] K. Wilken and J. Shen, "Concurrent Error Detection Using Signature Monitoring and Encryption," *Proc. Int'l Working Conf. Dependable Computing for Critical Applications*, pp. 365-384, 1991.



Kent Wilken (M'84) received the BS and MS degrees in electrical engineering (with distinction) from Stanford University, Stanford, California, in 1977, and the PhD in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, Pennsylvania, in 1990.

From 1977 to 1985, he was with the Hewlett Packard Company, where he was involved in disk controller architecture, mass storage performance, I/O architecture, and fault-tolerant computing. In 1990, he joined the Department of Electrical and Computer Engineering of the University of California, Davis, where he is currently an assistant professor. His research interests include compiler optimization techniques and architecture techniques for high performance computing and for dependable computing.

Dr. Wilken is a member of the IEEE and the ACM.

Timothy Kong received the BS degree in electrical engineering from the University of California, Berkeley, in 1989, and the MS degree in electrical and computer engineering from the University of California, Davis, in 1991. He is currently a PhD candidate in electrical and computer engineering at the University of California, Davis. He has held internship positions at Silicon Graphics and at Microsoft, and has worked as a UNIX system administrator at the University of California, Davis. His research interests include compiler optimization, computer architecture, and fault-tolerant computing.