Software-Based Concurrent Control Flow Checking

Volkmar Sieh and Joachim Hönig
University of Erlangen–Nürnberg, Germany *

Email: sieh@informatik.uni-erlangen.de

Abstract

A software technique to concurrently detect errors is presented. It is based on monitoring the sequence of high-level language statements using assigned signature control flow checking. A preprocessor analyzes the program source and inserts control flow checking code into the program. This method is shown to have only a low execution time overhead and moderate program size overhead. It was evaluated on different architectures. Error coverage and latency were measured using a fault injector based on the UNIX `ptrace` system call.

Keywords: (Concurrent) Error Detection, Control Flow Checking, Fault Injection Experiments, Signature Analysis, Watchdog Processors.

1 Introduction

Fault-tolerant, reliable computing requires effective methods to concurrently detect errors caused by hardware malfunctions. On the system level, many of these errors are manifested as disturbances of program control flow [9]. Therefore, many hardware methods for control flow checking (CFC) have been devised, using a simple coprocessor as watchdog processor to concurrently monitor the run-time program flow of the main processor. A survey of this and other watchdog-based error detection techniques is given in [4].

In the derived signature CFC approach program flow information is compacted into signatures and embedded within the program code. At run-time, a simple watchdog processor monitors the address and data lines. Using the signatures encountered in the instruction stream, correct program flow can be validated [14, 12, 2, 10, 13, 15, 11]. Unless the checker is included on the processor chip itself, this approach does not agree with internal instruction caches, because small loops may execute entirely within the cache. Other methods (assigned

*This research was supported by DFG (Deutsche Forschungsgemeinschaft) as part of SFB 182

signature CFC) [3, 6, 8] do not suffer from this problem. Instead of monitoring each instruction, the watchdog processor only checks the correct sequence of signatures transferred explicitly upon entering or leaving a program segment. On the other hand, small program flow disturbances, like skipping a single instruction, cannot be detected in this approach.

Our method is a software implementation of the watchdog processor of the assigned signature CFC method called extended structural integrity checking (ESIC) [6]. This method added some improvements and extensions to the the original SIC method presented in [3]. Before the compilation stage, a preprocessor automatically creates a self-checking program from a given program source. In our implementation the preprocessor is built to accommodate the 'C' programming language, but the implementation can be adapted to any other procedural programming language. As this approach does not depend on processor architecture, we were able to evaluate this technique with fault injection experiments on several different systems.

As this approach is implemented in software, it is a low-cost alternative to watchdog processors in a wide variety of systems and architectures. Concurrent error detection techniques based solely on software methods are rare. Software-implemented CFC is presented in [16]. The BSSC (Block Signature Self Checking) method of [7] is comparable to our approach, but our method is capable of correctly handling interrupts and destination addresses calculated during program execution.

The next section 2 gives a brief description of the ESIC method. Section 3 contains a short overview of the implementation and design of the preprocessor. Our checking algorithm is presented in section 4. An example of a modified program source is given in section 5. Section 6 presents some experimental results obtained from fault injection experiments performed on different processor architectures.

2 Control Flow Checking Using ESIC

In the ESIC technique a preprocessor analyzes the program flow of a high-level language source and builds the control flow graph (CFG). Each vertex of the CFG represents a sequence of statements, and each edge the transfer of control to another statement sequence. A unique signature is assigned to each vertex. The preprocessor produces two outputs: a modified program source text, including statements to transfer the signatures to a watchdog processor, and a tabular representation of the CFG, used by the watchdog processor to monitor run-time execution. Whenever a transfer of control not represented by an edge in the CFG is detected, the watchdog processor raises an exception.

In contrast to the SIC method [3], ESIC is capable of handling interrupts and function pointers correctly. Many programming languages allow the call of functions by function pointers, so that all possible program flows cannot be extracted by mere syntactical analysis. The same problem arises if spontaneous control flow transitions due to interrupt handling are allowed. Thus, in the ESIC method, not the entire CFG is built, but only a set of subgraphs, with each subroutine (function declaration) mapped to a different subgraph CFG^f, where f

is a number uniquely identifying this function. Function calls are *not* included into the CFG. The signatures generated for a vertex are formed by the vertex number and the function number f. The watchdog processor not only checks the vertex number within CFG^f, but also monitors the function number. The function number may only change if a function is called, that is, control flow switches to a different subgraph starting from its initial vertex. Any other branch into another subgraph is rejected. Upon encountering the initial vertex of a subgraph, the previous signature and function ID are pushed onto the watchdog processor stack. To resume checking of the calling program, upon end of a function (or upon encountering a `return` statement) the last signature and function ID are restored. One of the drawbacks of this method is the possibly very high error latency, if the program contains recursive calls (errors are possibly detected only after the final function return).

Using ESIC, our idea is to merge both watchdog and main program into one program executed by the processor itself. Instead of the two-way strategy, the preprocessor only generates a self-checking program from a given source.

3 The Preprocessor

The preprocessor was implemented using the `bison` parser generator. The parser was designed to recognize the control structures of the C programming language down to statement level. From standard building blocks, such as simple statements, loops and conditional statements, the sub-CFGs are constructed following a bottom-up strategy. A flow checking code sequence is inserted at the appropriate places in the control structures, and after every sf-th statement. The parameter sf (statement factor) defines the maximum number of simple statements (statements performing no branches) to form a statement sequence (vertex in the CFG). Low values of SF increase error coverage, whereas larger factors generate code with less overhead. With a different flag, a reduction algorithm [5] can be activated. With reduction turned on, some vertices (small branches) are removed, but only if the maximum statement sequence length is not exceeded. Cycles in the CFG (corresponding to program loops) are not allowed to be removed completely. Thus, each loop contains flow checking code at least once, avoiding high error latencies that might occur otherwise.

The following three types of vertices are handled differently by the preprocessor:

Start of function (SOF). The initial vertex of a function. Most programming languages allow execution of a function (subroutine) only from a single entry point. A SOF vertex does not have any incoming edges.

End of function (EOF). Statement sequences of this type have no outgoing edges. They occur either at the end of a function or contain a `return` statement.

Standard vertex. All other vertices are standard vertices.

Fig. 1 shows a small program fragment and its CFG. The parameters are statement factor

Figure 1: Example program and CFG

4 Checking Algorithm

In order to facilitate quick flow checking of both the function number f and the succession of vertices, the individual subgraphs $CFG^f = (V^f, E^f)$ of each function are represented by adjacency matrices (control flow matrices, CFM^f). The elements cfm^f_{ij} of these matrices are defined as follows:

$$\forall v^f_i, v^f_j \in V^f \wedge (v^f_i, v^f_j) \in E^f : cfm^f_{i,j} = f$$
$$\forall v^f_i, v^f_j \in V^f \wedge (v^f_i, v^f_j) \notin E^f : cfm^f_{i,j} = 0$$

The checking code always accesses the CFM of the presently running function f. If, in a **standard vertex** v^f_j, the program flow from the previous vertex v^f_i was correct, there is a corresponding edge in E^f. Therefore, a lookup of the element $cfm^f_{i,j}$ will yield a value equal to f. If the value retrieved is equal to 0, an illegal transition within the same function has occurred. If the value is not equal to either f or 0, an illegal transition from a different function has happened. CFM lookups take advantage of the processor cache.

Only the ID of the presently running function, used for comparison with the value retrieved from the CFM, is kept in a global variable; all other variables are local and destroyed automatically upon returning from a function. Thus, explicit stack operations at SOF and EOF vertices are not necessary:

Upon encountering the initial **SOF vertex** of a function f, the ID (number) of the previous (calling) function is saved and the global variable is updated. As functions are allowed to be called anytime, no checks are performed. The initial vertex is always numbered 0.

Upon reaching an **EOF vertex**, correct program flow is checked the same way as standard vertices are. Afterwards the previous function ID is restored to the global variable.

5 Example

The preprocessor output of the example program (Fig. 1) is shown in Fig. 2.

```
example()           /* 3rd function */
{
    /* v0 */ static unsigned char __cfm[5][5] = {
        0, 0, 0, 0, 0,
        3, 0, 0, 3, 0,
        0, 3, 0, 0, 0,
        0, 3, 3, 0, 0,
        3, 0, 0, 3, 0,
    };
    unsigned register __sign = 0;
    unsigned char   __fn_sav = __fnum;
    __fnum = 3;
    while (a < 10) {
        /* v1 */ if (__cfm[1][__sign] ≠ __fnum) __cfc_err();
        __sign = 1;
        if (b == 0) {
            /* v2 */ if (__cfm[2][__sign] ≠ __fnum) __cfc_err();
            __sign = 2;
            c = a;
        }
        /* v3 */ if (__cfm[3][__sign] ≠ __fnum) __cfc_err();
        __sign = 3;
        a++;
    }
    /* v4 */ if (__cfm[4][__sign] ≠ __fnum) __cfc_err();
    __fnum = __fn_sav;
}
```

Figure 2: Preprocessor Output

At the first vertex v_0 (SOF) of the function `example()`, the preprocessor inserts code to declare and initialize the CFM. Within each access into the CFM the previous signature is used as the column number, whereas the active signature is the row number. Function IDs are assigned in sequential order; if we assume that `example()` was declared third in the program, the function ID 3 will be assigned. But before the function ID (global variable `__fnum`) is set to 3, the calling function number is stored in a local variable, to be restored at the EOF vertex v_4.

The code inserted at the standard vertices v_1, v_2 and v_3 reads an element from the CFM. Upon errors, the external function `cfc_err()` is called to terminate the program.

The code at vertex v_4 (EOF) restores, if no error is detected, the function ID of the calling function, saved at the beginning (SOF vertex v_0) of `example()`.

6 Experimental Evaluation

In order to evaluate error coverage, error latency and overhead, different programs were submitted to tests on different systems. As the checking method is based on assigned signa-

ture control flow checking, we expected the error coverage to be equal or even higher than the error coverage of a comparable watchdog processor, because even errors not disturbing control flow directly (for example errors in general purpose registers) can be detected in our method.

In order to acquire meaningful runtime overhead measurements, CPU intensive programs were used as benchmarks, though the results are less flattering than the I/O-intensive applications used in some other publications. Thus, test programs were slightly modified dhrystone (integer operations) and whetstone (floating point arithmetic) benchmarks, a multigrid program to solve the two-dimensional Poisson equation, and a multigrid program to solve the Navier-Stokes equation. The architectures used were Sparc (Sun 4/75), MC68020 (Sun 3/60), MC88100 (MVME 188), and 80386/486 (Sequent Symmetry). The same compiler (`gcc -V2.2.2`) was used in all cases. All available code optimization algorithms were employed (`-O2` option). As it turned out, optimization is non-flattering as well. Though overall runtime was reduced, relative runtime overhead increased on all architectures.

6.1 Overhead

The insertion of self-checking code into the program source causes overhead both in program size and in runtime. There is a quadratic increase in CFM size with the length of functions. Structured, modular programs will therefore cost less in terms of program size overhead. Furthermore, size overhead depends on the granularity of the checks, which is specified both by the statement factor (1, 2, 5 and 10 in our experiments, subsequently referred to as SF-1, SF-2, SF-5 and SF-10), and on whether reduction is enabled or not (referred to as RF-2, RF-5 and RF-10 for statement factors 2, 5 and 10. With statement factor 1 no reduction is possible). With some programs the object file length was more than tripled with SF-1. Yet, after linking with unchecked startup code and libraries, the size of the executable program never increased more than about 70%. If the data space allocated during execution is taken into account, the total storage overhead for the Poisson program (3.5 MByte data space) is about 0.5%.

Runtime overhead depends on the statement factor as well. Values are in the range of 30–39% (Dhrystone Sparc), 42–107% (Whetstone Sparc), 22–57% (Poisson Sparc) and 3–8% (Navier Stokes Sparc). Of all benchmarks the Poisson program covers midrange in terms of runtime overhead. So in Fig. 3 the runtime overhead for this program for all processors in the test is shown.

Both CISC processors (68020 and i486) show remarkably less runtime overhead. This is mainly due to the fact that both systems were not equipped with a floating point coprocessor. Therefore, a considerable amount of CPU-time is spent on math emulation. As the granularity of checks is measured in high-level language statements, there is a higher number of machine instructions per signature on these systems.

Figure 3: Runtime Overhead (Poisson)

6.2 Fault Injection

For error coverage and latency measurements a fault injector based on the UNIX `ptrace` interface was used. Originally intended for debugging purposes, this interface allows access to the address space of the target program and to register contents. No fault can be injected into the state of external devices or into the operating system. Using this interface we were able to inject faults into a running process without modifying the program code.

For our tests we decided to inject only single-bit faults into the processor registers. With this simple fault model the fault injector can be ported to many different systems with only little or no change. We did not inject faults into memory, as these faults either have no effect, or can be detected with very high probability using error detecting codes. Test runs were made for single-bit faults injected into the program counter, into the stack pointer or into a random other register.

Most systems provide standard error detection methods like monitoring memory access rights, which, as our results show, cover a substantial amount of errors. Therefore, instead of measuring the overall fault coverage of our method, we were interested in the increase of coverage achieved by additionally employing our method. After injecting a fault, the following reactions were observed:

- System Error: One of the standard, built-in system error detection methods responded, and the process died because of a signal. Examples are segmentation violation, word alignment error, illegal instruction, and floating point exceptions.
- CFC-Error: A self-checking program detected a control flow error and terminated.

Figure 4: Frequency of different observations (Poisson Sparc)

For the PC test, the overwhelming majority of errors were detected by system methods, mostly segmentation violations. As errors were injected into a random bit position, it is no surprise that the percentage of segmentation violations is about 50%, as only half of the address bits are used in an address space of 2^{32} for a 64 kByte program. Depending on program size and data space size, the percentage of segmentation violations may vary substantially. In the register test, the majority of faults do not cause an error. Though only very few register errors are detected by control flow checking (1.3%), the vulnerability to register errors is typically drastically reduced. As the registers are used for the control flow checks as well, they are more frequently reloaded.

Figure 5: Fault Coverage (Poisson)

The diagrams show that for many processor architectures the number of errors not detected by standard methods was significantly reduced. The tests on the 88100 and the i486 system did especially well: undetected errors were reduced from 25% to 13% and 11% to 4%, respectively (comparison of the unmodified program (Orig) with SF-1).

As part of the experiments error latency was measured by the fault injector. Unfortunately, most UNIX systems have a clock resolution of 1/60 second, so it is difficult to measure time intervals in the range of 0 to 10 milliseconds. We switched to counting instruction cycles between injection of a fault and detection of an error in single step mode.

The mean error latency time is shown in Table 1 for the dhrystone benchmark. The values of the standard detection methods (System), and of the standard detection methods combined with control flow checking (CFC) are given as the mean number of instruction cycles until the error was detected. The statement factor in these experiments was SF-1, values are shown for program counter errors (PC), register errors (Reg) and stack pointer errors (SP). The results show that the latency of the control flow checks is in the same order of magnitude as the system error detection methods. In our experiments we could not find a dependency between latency and statement factor.

	PC	Reg	SP
System	38	443	52
with CFC	29	523	156

Table 1: Mean Error Latency (Dhrystone, Sun 3/60, number of instructions)

7 Conclusion

The number of undetected errors can be significantly reduced by using the control flow self checking method proposed. Depending on the parameters (statement factor, reduction) and on the program used, the overhead varies between 0-70% (program size overhead) and 10-100% (time overhead). By changing the preprocessor parameters, overhead and error coverage can be adjusted to given dependability requirements. Manually changing the source code is not necessary.

The method can be ported to any system. We intend to use this approach in the MEMSY [1] multiprocessor system with fault tolerance, which is based on MVME188 processing nodes. With some refinement, like modifying the compiler instead of using a preprocessor, and with some fine tuning to accommodate the 88100 processor, we believe it will be possible to achieve even better results.

References

[1] M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand, J. Hönig, W. Hohl, E. Michel, and A. Pataricza. Fault Tolerance in Distributed Shared-Memory Multiprocessors. In A. Bode and M. Dal Cin, editors, *Parallel Computer Architectures*, number 732 in LNCS, pages 31–48. Springer, 1993.

[2] J. B. Eifert and J. P. Shen. Processor monitoring using asynchronous signatured instruction streams. In *Proceedings 14th FTCS*, pages 394–399, 1984.

[3] D. J. Lu. Watchdog processors and structural integrity checking. *IEEE Transactions on Computers*, C-31(7):681–685, July 1982.

[4] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors - a survey. *IEEE Transactions on Computers*, 37(2):126–137, February 1988.

[5] E. Michel. *Fehlererkennung mit Überwachungsrechnern in Multiprozessorsystemen.* Arbeitsberichte des IMMD 25(6), Universität Erlangen-Nürnberg, 1992.

[6] E. Michel and W. Hohl. Concurrent error detection using watchdog processors in the multiprocessor system MEMSY. In *Proceedings of the 5th Int. GI/ITG/GMA Conference on Fault Tolerant Computing Systems*, volume 283 of *IFB*, pages 54–64. Springer, 1991.

[7] G. Miremadi, J. Karlsson, U. Gunneflo, and J. Torin. Two software techniques for on-line error detection. In *Proceedings 22nd FTCS*, pages 328–335. IEEE, 1992.

[8] A. Pataricza, I. Majzik, W. Hohl, and J. Hönig. Watchdog processors in parallel systems. *Microprocessing and Microprogramming*, 39:69–74, 1993.

[9] M. Schmid, R. Trapp, A. Davidoff, and G. Masson. Upset exposure by means of abstraction verification. In *Proceedings 12th FTCS*, pages 237–244, 1982.

[10] M. A. Schuette and J. P. Shen. Processor control flow monitoring using signatured instruction streams. *IEEE Transactions on Computers*, 36(3):264–276, March 1987.

[11] M. A. Schuette and J. P. Shen. Exploiting instruction-level resource parallelism for transparent, integrated control-flow monitoring. In *Proceedings 21st FTCS*, pages 318–325, 1991.

[12] J. P. Shen and M. A. Schuette. On-line self monitoring using signatured instruction streams. In *1983 IEEE International Test Conference*, pages 275–282, 1983.

[13] J. Sosnowski. Detection of control flow errors using signature and checking instructions. In *Proc. 18th IEEE ITC*, pages 81–88, 1988.

[14] T. Sridhar and S. M. Thatte. Concurrent checking of program flow in VLSI processors. In *1982 IEEE International Test Conference*, pages 191–199, 1982.

[15] K. D. Wilken and J. P. Shen. Concurrent error detection using signature monitoring and encryption. In *1st International Working Conference on Dependable Computers in Critical Applications*. Springer, Wien, 1990.

[16] S. S. Yau and F. Chen. An approach to concurrent control flow checking. *IEEE Transactions on Software Engineering*, SE-6(2):126–137, March 1980.