Z
699
C3
no. 91-78

Fault Tolerance in Super-Scalar and VLIW Processsors

Douglas M. Blough*
Alexandru Nicolau

*Department of Electrical and Computer Engineering
Department of Information and Computer Science

University of California, Irvine

Irvine, California 92717

Technical Report No. 91-78

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Fault Tolerance in Super-Scalar and VLIW Processors*

Douglas M. Blough
Department of Electrical and Computer Engineering
University of California
Irvine, CA 92717

Alexandru Nicolau
Department of Information and Computer Science
University of California
Irvine, CA 92717

Abstract

In this paper, we present a method for utilizing the spare capacity in super-scalar and very long instruction word (VLIW) processors to tolerate functional unit failures. Unlike previous work that was primarily interested in detection of transient faults, we are concerned with more permanent and/or intermittent faults which necessitate processor reconfiguration. Our method utilizes the VLIW compiler or the super-scalar scheduler to insert redundant operations whenever idle functional units exist. The results of these redundant operations are used to detect and diagnose functional unit failures. For super-scalar processors, the scheduler can then utilize this information to ensure that operations are performed only on non-faulty units. In VLIW processors, this is equivalent to recompiling the code to run on the remaining non-faulty functional units. Since in certain applications, recompilation may not be possible, we consider two alternative reconfiguration strategies for VLIW processors. These strategies sacrifice storage space and execution time, respectively, in order to reconfigure without recompiling. We present Markov models that describe the behavior of processors using these different approaches and we evaluate their reliabilities. The results show that, while super-scalar and VLIW with recompilation provide the highest reliability, all proposed strategies significantly increase reliability over that of an unprotected processor.

*This work was supported in part by NSF under Grant CCR-9010547.

1 Introduction

One of the principal uses of fault-tolerant computer systems is in applications where repair is difficult or impossible. These are typically applications in which systems are placed in remote areas, *e.g.* underground, underwater, or in space. Failures of such systems are extremely costly to repair and may be fatal to the system. In addition to requiring ultra-high reliability, many of these applications demand high performance. In this paper, we present a fault tolerance approach that is specifically designed for a large class of high-performance processors, namely those that utilize multiple functional units to exploit instruction-level parallelism. Our approach is capable of significantly increasing the reliability of such processors with only minor hardware modification and little overhead.

Dependences in a sequential program require that a parallel program which is a direct translation of the sequential program maintain the order of execution of some instructions. Instructions which are independent can be executed in any order, or simultaneously. A popular form of fine-grain parallelism is to use multiple functional units to execute independent instructions from a sequential program simultaneously. Two main approaches to this instruction-level parallelism have been developed. In the first approach [3, 7], operations are statically scheduled on the multiple functional units by the compiler. Processors utilizing this approach, known as very long instruction word (VLIW) processors, allow the maximum amount of parallelism to be extracted from a sequential program since the compiler can perform global dependence analysis. In the second approach [9], operations are scheduled dynamically on the functional units in such a way as to maintain a correct ordering. This approach, referred to as super-scalar, is simpler than VLIW since branches can be handled easily at run time but in general it will not extract as much parallelism since only local independences are exploited.

Both of these approaches to instruction-level parallelism attempt, in each clock cycle, to issue as many operations as there are functional units. This ideal situation is not possible when dependences exist in the sequential program. This means that some functional units will be idle at various points in the computation. This is illustrated by the data shown in Table 1. These data, collected using the percolation scheduling compiler described in [13], represent an average taken across the Livermore Loops, the Stanford Benchmarks, and several programs in computational fluid dynamics. The data were calculated assuming that all instructions take one cycle to complete. The table clearly indicates that a significant amount of unused capacity exists. The dynamic NOP percentage is the percentage of NOP's that occur during execution and it ranges from 9% when there are two functional units to 57% when the number of functional units is 16. Clearly, the higher the performance, the more spare capacity there is to exploit for fault tolerance purposes. These data

Number of Functional Units	Average Speed-Up	Dynamic NOP Percentage (Approx.)
2	1.82	9%
4	3.09	23%
8	4.90	39%
16	6.81	57%

Table 1: Speed-Up and NOP Percentage for Percolation-Based Compiler of [13]

are fairly conservative in the sense that even higher NOP percentages were reported on benchmarks done on the Multiflow TRACE 14/300 in [14].

Our approach to fault tolerance utilizes these idle functional units to execute redundant operations. A redundant operation is a replica of a primary operation executed on another unit. These operations are begun during the same clock cycle and their results are compared when the operations are completed. This duplication and comparison will detect any fault within the primary or redundant functional unit that affects the operation being performed. A sequence of such comparisons is used to isolate a fault to a particular unit. Once a faulty unit has been identified, no further operations should be scheduled on it. In super-scalar processors, this is not difficult since the scheduling is dynamic. In VLIW, a more involved reconfiguration procedure is necessary. In Section 3, we discuss our fault tolerance approach in more detail and we outline several alternative reconfiguration strategies for VLIW processors.

In general, the set of functional unit failures that can be tolerated by our approach depends on the way in which units are grouped for comparison purposes, on the diagnosis mechanism utilized, and on the reconfiguration procedure. In Section 4, we present Markov models for processors that utilize our proposed reconfiguration strategies. Using these models, we evaluate the reliabilities of the proposed strategies and compare them to each other as well as to a non-fault-tolerant processor. The results show that significant increases in reliability can be obtained by utilizing our approach.

Before presenting these results, we discuss related work.

2 Related Work

Almost all of the previous work in the area of multi-functional unit processors has been concerned with optimizing performance [7, 9, 11]. This work attempts to reduce the unused capacity in such a processor by keeping each functional unit as busy as possible. As was shown in Table 1, state-of-the-art scheduling techniques still result in a significant amount of unused capacity in the processor. The first research to consider utilizing this spare capacity for fault tolerance purposes was [14]. In that work, the authors proposed using idle functional units to detect transient faults through control-flow monitoring. This significant paper showed that 99% of control-flow errors can be detected on the Multiflow TRACE [3] computer with negligible overhead. For transient faults, detection is essential while diagnosis and reconfiguration are not necessary since there is no physical damage to the processor.

The idea of utilizing spare capacity in multiprocessor/multicomputer systems to execute redundant tasks was suggested in [1, 4, 6, 8]. In [6], the authors proposed using idle nodes in a distributed system to execute redundant tasks. The scheduling of these tasks is done by software and the goal is to replicate all critical tasks so as to ensure their correct completion. The problems of diagnosis and reconfiguration are not considered. In [4], spare capacity in a bus-based multiprocessor system is used to execute comparison tasks that form the basis of a diagnosis strategy [5, 2, 10]. Here, the scheduling and diagnosis are done at a high level in the system by software. In [1, 8], the problem of scheduling the individual tasks of a larger computation in a fault-secure manner is considered. This requires, as a minimum, replication of each task in the computation, something which is in most cases not possible using spare capacity alone and hence, this approach causes decreased performance. In addition, the approach is concerned only with fault detection and does not consider how to diagnose faulty processors or reconfigure the system.

We are concerned herein with permanent or intermittent faults in one or more units of a multi-functional unit processor. Without fault tolerance, such a processor would be rendered inoperable by even one such fault. In our approach, the unused capacity in such a processor can be utilized to detect and diagnose these faults. Unlike in [1, 4, 6, 8], detection and diagnosis must be accomplished in hardware. We present simple modifications to the design of a multi-functional unit processor that enable this hardware detection and diagnosis to take place. Whereas with transient faults reconfiguration is not an issue [14], for permanent and intermittent faults the processor must be reconfigured so that it can be used after one or more functional unit failures. We will present and evaluate multiple reconfiguration alternatives, some accomplished solely in software and others requiring hardware assistance. This work is particularly valuable for long-life applications since it can significantly extend the operational life of the processor.

3 Fault Tolerance Approach

In this section, we propose an active redundancy approach to the fault tolerance of multi-functional unit processors. The approach can be divided into fault detection, fault diagnosis, and reconfiguration components. These components are discussed in turn in the remainder of the section.

3.1 Fault Detection

The basic means of fault detection that we utilize is duplication and comparison. A functional unit that would otherwise be idle executes an operation identical to that executed by a busy functional unit and the results of these operations are compared. A mismatch in results indicates a fault in one of the two functional units. For this purpose, we partition the functional units in the processor into groups of identical units. These groups are referred to as *comparison groups* since any two processors in the same such group can execute duplicate operations and compare results. To accomplish this, the hardware must be capable of delivering the outputs of any pair of units in the same comparison group to a comparator circuit. Since a switching circuit that performs this task is quite complex if the number of units in a group (referred to as the *group size*) is large, the group size should be small. For diagnosis purposes, the group size must be at least three. Since the total number of functional units is typically a power of two, we utilize a group size of four.

Our duplicate and compare strategy requires a mechanism for scheduling the redundant operations, hardware for selecting and comparing outputs, and a method for controlling the selection and comparison. The mechanism for scheduling redundant operations differs depending on whether the processor is a VLIW or super-scalar processor. In a VLIW processor, these operations are scheduled by the compiler whenever there is a NOP present in the code. In a super-scalar processor with dynamic scheduling, the scheduler must be modified to issue duplicate instructions whenever there are at least two idle units from the same group. In order to control the selection and comparison hardware, each functional unit should have two additional control bits indicating whether the current operation is a primary or redundant operation and to which unit the result should be compared. For this purpose, the units in a group are numbered from 0 to 3. If the control bits inside a unit contain the unit's own ID, this indicates the current operation is a primary operation. Otherwise, the control bits give the ID of the unit to which the result should be compared. In this case, the write-back stage of the pipeline in the unit should be disabled. In a VLIW processor, these control bits are generated by the compiler and stored as additional bits for each operation. This incurs a slight storage overhead. In a super-scalar processor, the bits are generated dynamically by

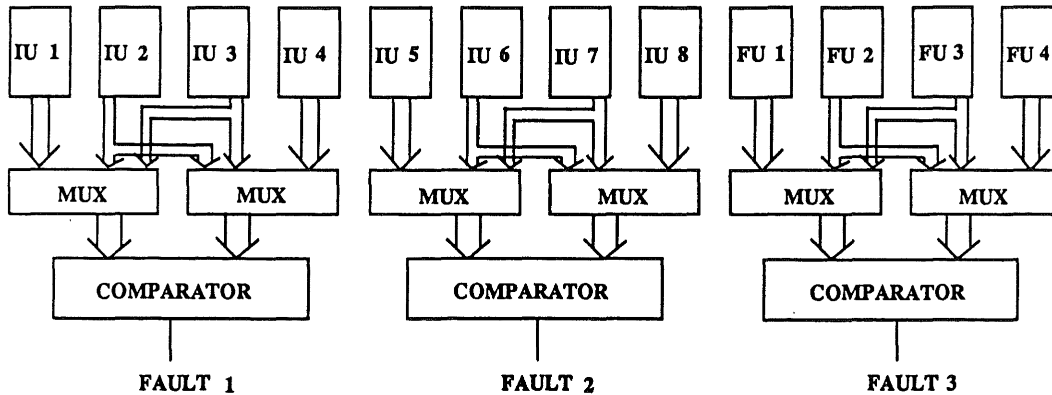


Figure 1: A 12-Unit Processor with Fault Detection Circuitry

the scheduler.

To simplify the hardware for selection and comparison, we execute at most one comparison per group during each cycle. Thus, only one comparator is needed for each comparison group. Figure 1 shows an example of a 12-unit processor with eight integer units and four floating point units. This processor has three comparison groups and hence, three comparators. The comparators are sequential circuits to prevent temporary incorrect values on the FAULT lines due to functional units' outputs that change at slightly different times. The switching circuit is quite simple requiring only two multiplexers to direct the units' outputs to the comparator. The control lines for these multiplexers can be obtained easily from the control bits described above. Such a processor is capable of executing three duplicate and compare tasks per cycle making efficient use of the spare capacity in the processor.

An important parameter of a fault detection mechanism is its detection latency. The detection latency is defined as the time between a fault's occurrence and its detection. To minimize the detection latency, each functional unit should participate in a duplicate and compare task as often as possible. This implies that these tasks should be scheduled in a round-robin fashion. Unfortunately, this is not possible in either a VLIW or super-scalar processor. In VLIW, the scheduling is done statically. For straight-line code, it is therefore possible to guarantee a round-robin schedule.

```
main()
{
    register int i;
    double x[1002], y[1002], z[1002];

    for (i = 2; i <= 998; i+ = 1) {
      x[i] = z[i] * (y[i] - x[i - 1]);
    }
}
```

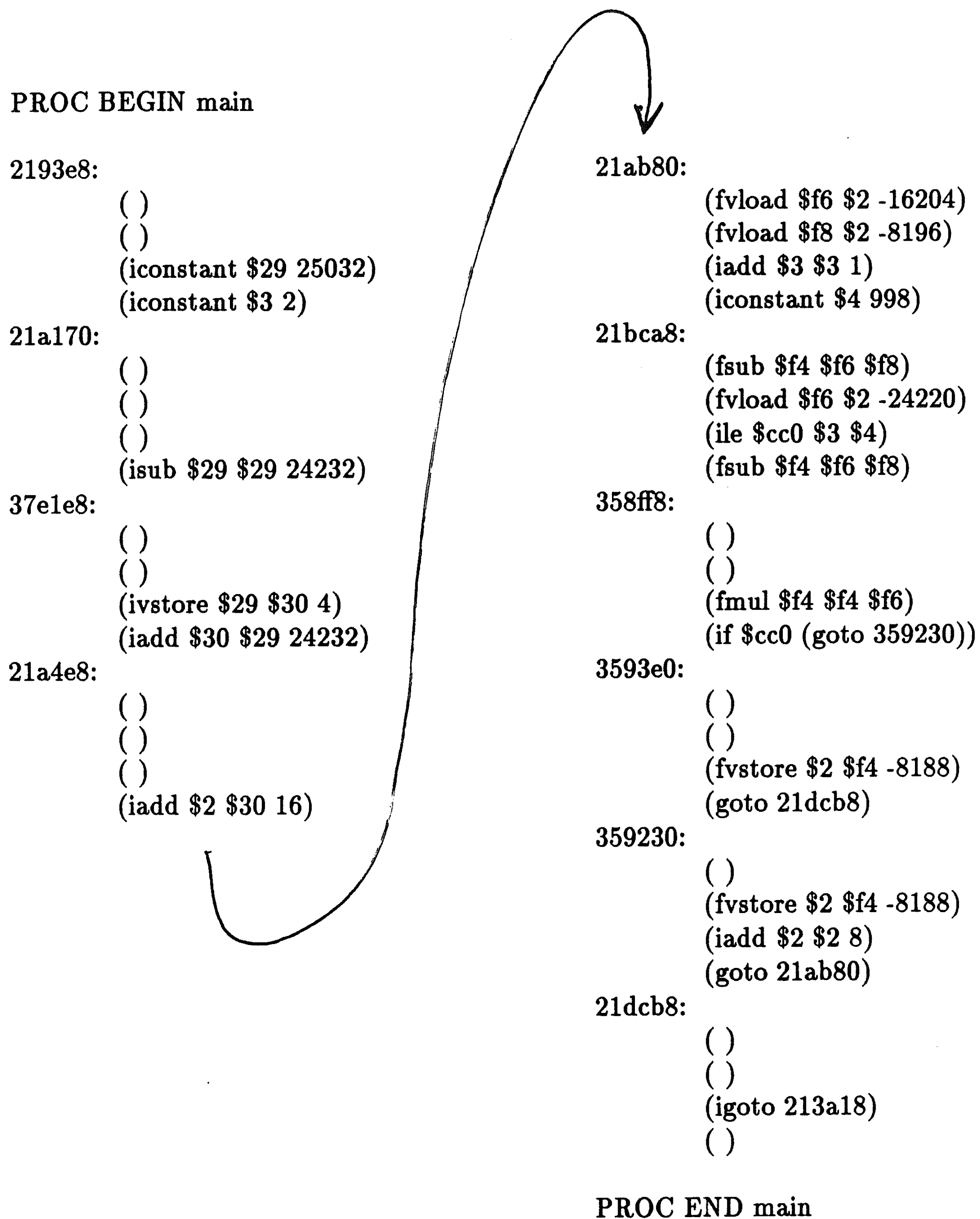
Figure 2: Source Code for Livermore Loop Number 5

However, in segments of code containing branches, the compiler cannot achieve this since it does not know how the dynamic execution will proceed. In the worst case, there may be a single NOP inside a loop that is executed many times. This means that the same pair of units will be compared repeatedly while other units may have faults that remain undetected until the loop is exited. One way of alleviating this problem is to use loop unwinding [12]. Loop unwinding has been used to expose parallelism. In this case, it can be used to increase the number of NOP's inside a loop so as to allow the compiler to institute a fair schedule. A negative side-effect of loop unwinding is that it increases the amount of storage required for the code but this is acceptable given that it may reduce the detection latency significantly.

Figures 2–5 show an example of the scheduling that is done on a VLIW machine. Figure 2 shows the C code for one of the Livermore loops. In Figure 3, the intermediate 3-address code generated by the GNU C compiler (gcc) for the loop of Figure 2 is shown. Next, the code is parallelized for a four functional unit VLIW machine. This parallelized code is shown in Figure 4. Each address label in this figure corresponds to an instruction containing four separate operations to be performed simultaneously on the functional units. The empty parentheses represent NOP's that can be used to execute redundant operations. Finally, Figure 5 shows the parallelized code with redundant operations inserted. Here, the four functional units form a single comparison group. As an example of a redundant operation, consider instruction 37e1e8. Here, the iadd operation is executed on both functional unit 1 and functional unit 3. The results of these two executions are then compared using the fault detection circuitry described previously. The ivstore operation in the same instruction can not be duplicated even though there is one more NOP in the instruction because there is only one comparator per group. Note that in the initialization portion of the program (instructions 2193e8 through 21a4e8) which is straight-line code, a round-robin schedule has been implemented. Inside the loop

```
    (PROC_BEGIN main
(LABEL main)
    (ICONSTANT $29 25032)        ; $29:= 25032
    (ISUB $29 $29 24232)         ; $29:= 800
    (IVSTORE 4 $29 $30)          ; M[804]:= $30
    (IADD $30 $29 24232)         ; $30:= 25032
    (ICONSTANT $3 2)             ; $3:= 2 (i=2)
    (IADD $2 $30 16)             ; $2:= 25048 (base for x[i],y[i],z[i])
(LABEL L5)
    (FVLOAD $f6 -16204 $2)       ; $f6:= M[8844] (y[i])
    (FVLOAD $f8 -8196 $2 )       ; $f8:= M[16852] (x[i-1])
    (FSUB $f4 $f6 $f8)           ; $f4:= y[i]-x[i-1]
    (FVLOAD $f6 -24220 $2)       ; $f6:= M[828] (z[i])
    (FMUL $f4 $f4 $f6)           ; $f4:= z[i]*(y[i]-x[i-1])
    (FVSTORE -8188 $2 $f4        ; M[16860]:= $f4 (x[i])
    (IADD $2 $2 8)               ; increment pointer
    (IADD $3 $3 1)               ; i+=1
    (ICONSTANT $4 998)
    (ILE $cc0 $3 $4)             ; i<=998 ???
    (IF $cc0 (LABEL L5))
    (IADD $8 $0 $30)             ; $8:= 25032
    (IVLOAD $30 -24228 $8)       ; $30:= M[804]
    (IADD $29 $0 $8)             ; $29:= 25032
    (IGOTO $31)
    (PROC_END main)
)
```

Figure 3: 3-address Code for Loop



```
PROC BEGIN main

2193e8:
        ( )
        ( )
        (iconstant $29 25032)
        (iconstant $3 2)
21a170:
        ( )
        ( )
        ( )
        (isub $29 $29 24232)
37e1e8:
        ( )
        ( )
        (ivstore $29 $30 4)
        (iadd $30 $29 24232)
21a4e8:
        ( )
        ( )
        ( )
        (iadd $2 $30 16)

21ab80:
        (fvload $f6 $2 -16204)
        (fvload $f8 $2 -8196)
        (iadd $3 $3 1)
        (iconstant $4 998)
21bca8:
        (fsub $f4 $f6 $f8)
        (fvload $f6 $2 -24220)
        (ile $cc0 $3 $4)
        (fsub $f4 $f6 $f8)
358ff8:
        ( )
        ( )
        (fmul $f4 $f4 $f6)
        (if $cc0 (goto 359230))
3593e0:
        ( )
        ( )
        (fvstore $2 $f4 -8188)
        (goto 21dcb8)
359230:
        ( )
        (fvstore $2 $f4 -8188)
        (iadd $2 $2 8)
        (goto 21ab80)
21dcb8:
        ( )
        ( )
        (igoto 213a18)
        ( )

PROC END main
```

Figure 4: Parallelized Code

```
PROC BEGIN main

2193e8:
        (iconstant $29 25032)
        (iconstant $29 25032)
        ( )
        (iconstant $3 2)
21a170:
        ( )
        ( )
        (isub $29 $29 24232)
        (isub $29 $29 24232)
37e1e8:
        (iadd $30 $29 24232)
        (ivstore $29 $30 4)
        (iadd $30 $29 24232)
        ( )
21a4e8:
        ( )
        (iadd $2 $30 16)
        ( )
        (iadd $2 $30 16)

21ab80:
        (fvload $f6 $2 -16204)
        (fvload $f8 $2 -8196)
        (iadd $3 $3 1)
        (iconstant $4 998)
21bca8:
        (fsub $f4 $f6 $f8)
        (fvload $f6 $2 -24220)
        (ile $cc0 $3 $4)
        (fsub $f4 $f6 $f8)
358ff8:
        ( )
        (fmul $f4 $f4 $f6)
        (fmul $f4 $f4 $f6)
        (if $cc0 (goto 359230))
3593e0:
        (fvstore $2 $f4 -8188)
        (fvstore $2 $f4 -8188)
        ( )
        (goto 21dcb8)
359230:
        (goto 21ab80)
        (fvstore $2 $f4 -8188)
        (iadd $2 $2 8)
        (iadd $2 $2 8)
21dcb8:
        (igoto 213a18)
        ( )
        (igoto 213a18)
        ( )

PROC END main
```

Figure 5: Parallelized Code with Redundant Operations

body (instructions 21ab80 through 359230), there are enough NOP's so that the compiler can implement a fair schedule, *i.e.* each functional unit executes at least one duplicated operation. Hence, in this case loop unwinding is not necessary to reduce the fault detection latency.

3.2 Fault Diagnosis

In our fault tolerance approach, fault diagnosis is done through analysis of a sequence of functional unit mismatches. With a group size of four, there are $\binom{4}{2} = 6$ pairs of units in each group. Our fault diagnosis hardware makes use of six latches, each corresponding to a different pair of units. Each time a pair of units produces a mismatch, the FAULT signal clocks a "1" into the corresponding latch. The correct latch can be determined based on the functional unit control bits described previously. If there is at most one faulty functional unit in a comparison group, the unit can be diagnosed as faulty after it has produced a mismatch with two other units. However, we prefer to diagnose a unit as faulty only when it has produced a mismatch with all other units in its comparison group. The circuitry required for this diagnosis is shown in Figure 6.

With only a single faulty unit, the diagnosis result will be the same whether a unit is diagnosed after two or three mismatches. The diagnostic latency of our approach will be greater than the alternative in this case since we must wait for one additional mismatch before producing the diagnosis result. The reason that we choose the slower procedure is because we are interested in diagnosing up to two faulty units per comparison group. If a unit is diagnosed as faulty after only two mismatches and there are nearly coincident faults in two units, then a non-faulty unit will produce two mismatches in a very short time. The first approach would incorrectly diagnose the non-faulty unit as faulty while our approach would not since the two non-faulty units will still match. Furthermore, the two faulty units will both produce mismatches with the two non-faulty units and they are likely to produce a mismatch with each other which will cause them both to be diagnosed as faulty. The only possible instance where two faulty units in a group cannot be diagnosed as faulty is when they fail in exactly the same way at almost exactly the same time. Even in this case, our approach is safe in that no non-faulty unit will be diagnosed as faulty.

It should be noted that we are concerned primarily with permanent and/or intermittent faults in this work so that once a unit has failed it is assumed to be of no use. The diagnosis mechanism described above utilizes this assumption. If the processor is to be used in an environment where transient faults due to environmental factors can occur, then the mechanism should be modified. If the mechanism is used

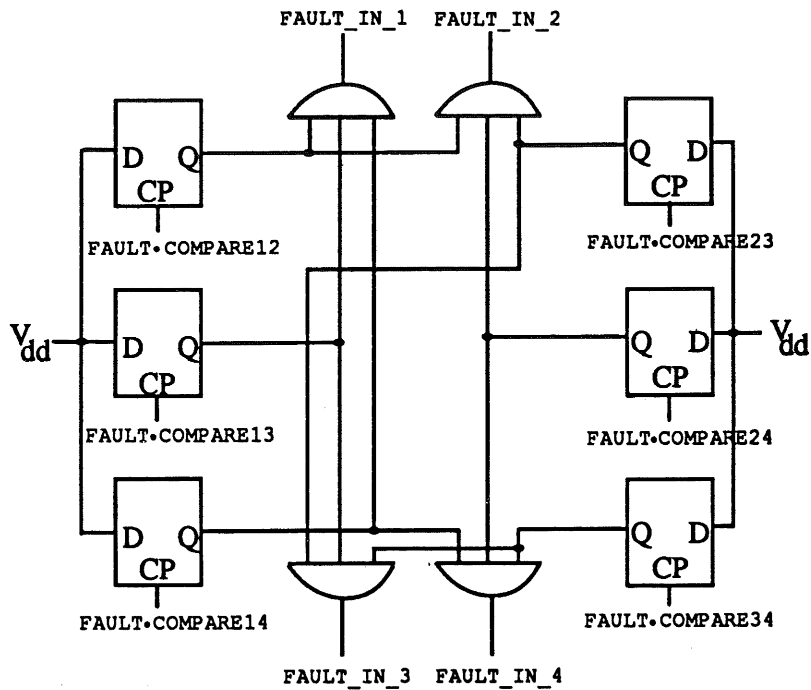


Figure 6: Diagnosis Circuitry for a Single Comparison Group

period of time may end up being diagnosed as faulty. A standard approach in this situation is to force units to produce a sufficient number of mismatches during a specified length of time before diagnosing them as faulty. This can be accomplished in our approach by adding a counter in front of each latch and triggering the latch off the desired bit of the counter. In order to prevent mismatches due to transient faults from building up over a long period of time, the counters should be reset periodically.

3.3 Reconfiguration

Since we are concerned with permanent and/or intermittent faults backward recovery strategies are not guaranteed to be successful. Hence, when faulty units are diagnosed we recover by reconfiguring the processor and restarting the computation. As stated in the previous section, our diagnosis mechanism is capable of diagnosing up to two faulty units per comparison group. Ideally, we would like to be able to reconfigure after each failure until the capability of the diagnosis mechanism is exceeded, *i.e.* until some comparison group has three faulty units. This maximum level of reconfiguration is possible for super-scalar processors since the scheduling is dynamic. The scheduler keeps track of the remaining non-faulty units and issues instructions only to those units. Since the scheduling in a VLIW processor is static, reconfiguration is not as easy. The most natural reconfiguration is accomplished by recompiling the code and inserting NOP's in every operation corresponding to a faulty unit. For this to work, there must be a remote machine available to recompile the code and download it into the VLIW processor. While this provides the maximum reconfiguration capability, it is slow and, in some applications, it is not possible. Hence, we consider two alternative reconfiguration strategies for VLIW processors.

One reconfiguration strategy is to store multiple versions of object code precompiled for different fault situations. The straightforward way of accomplishing this would store $n+1$ versions of object code to tolerate a single faulty unit in a processor with n functional units. To tolerate k faulty units in the processor would require $\binom{n}{k} + \cdots + \binom{n}{1} + \binom{n}{0}$ versions of the object code. Even for small values of n the storage space required for this is prohibitive when $k = 2$. An alternative is to store one version of the object code for each number of faulty units, *i.e.* one for the single fault case, one for the double fault case, *etc.* Although this greatly reduces storage requirements, the execution of the code requires hardware support in order to direct the operations into the correct functional units. This is feasible only for small values of k. Hence, this alternative, which we refer to as multiple code versions, provides less than the maximum reconfiguration capability.

Technique	Advantages	Disadvantages
Recompilation	maximum reconfiguration capability, no storage overhead, maximum performance	slow reconfiguration, not possible in all applications
Multiple code versions	fast reconfiguration, maximum performance	high storage overhead, low reconfiguration capability
Folding	fast reconfiguration, no storage overhead	poor performance, low reconfiguration capability

Table 2: Advantages and Disadvantages of VLIW Reconfiguration Alternatives

It becomes more difficult to implement multiple code versions when different types of functional units are present in the processor.

Another alternative which can be used to reconfigure after a single failure is to split each long instruction into two instructions of half the length. After one failure, either the right half of the functional units or the left half of the functional units will still be non-faulty. The appropriate half can then be used to execute the shorter instructions in twice the time of the original execution. This technique can be used when different types of functional units are present so long as the two halves are identically configured. We refer to this technique as folding. Folding can be applied multiple times but the execution time increases exponentially with the number of folds and the complexity of the hardware support also increases. Hence, from a practical standpoint, this technique can be used to tolerate only a small number of faulty units.

The advantages and disadvantages of these three reconfiguration alternatives for VLIW processors are summarized in Table 2. Note that the three alternatives sacrifice different resources in order to reconfigure. Recompilation uses reconfiguration time, multiple code versions sacrifice storage space, and folding sacrifices execution time. In the next section, we present and analyze detailed models for the reliability of processors that utilize one of these alternative reconfiguration schemes.

It is worth noting that in the specified approach, if a comparison group contains functional units of the same type and the group size is four, the number of functional units of each type in the processor must be divisible by four. For processors that have

only one type of functional unit this is not a problem. However, for processors with both integer and floating point units where the number of units of each type is not divisible by four, it may be necessary to have a hybrid comparison group containing units of both types. Duplication and comparison within hybrid groups is slightly more complicated. First, only integer operations can be used when comparing an integer unit and a floating point unit. Next, since floating point units will take more cycles to complete an operation than integer units, if a comparison is made between units of different type, the results of the integer unit will have to be stored until the floating point result is available. In addition, the floating point output will have to be converted to integer format before the comparison is done. Finally, there should be at least two floating point units in the hybrid group so that comparison of floating point operations can be done. If a floating point unit is only checked when it is executing integer operations, some faults in the unit could go undetected.

4 Reliability Analysis

In this section, we evaluate the reliabilities of the different reconfiguration strategies described in the previous section.

4.1 Markov Models

For the purposes of analysis, we separate the failure rate of a VLIW or super-scalar processor into multiple parts. We denote the failure rate of each functional unit in such a processor by λ_u and the failure rate of the remainder of the processor (everything other than the functional units) by λ_{oth}. Assuming exponentially distributed failure times, the overall failure rate for a fault-intolerant processor with n functional units is then $n \cdot \lambda_u + \lambda_{\mathrm{oth}}$. In general, the functional units are the most complex components of the processor and require the most area. Another component that occupies a large area is the register file. Due to this, we strongly recommend that a processor designed with our fault tolerance approach utilize error-correcting codes in the register file in order to reduce λ_{oth}. The remaining circuitry in the processor is quite simple relative to the functional unit complexity and consists mainly of an address decoder for the register file, a small amount of control circuitry, and an instruction scheduler (only in super-scalar processors). The bulk of the processor control circuitry is contained in the functional units which are usually fully pipelined processing elements. It is reasonable to assume that λ_{oth} is no greater than λ_u and will often (particularly for VLIW processors) be much smaller.

The Markov model for a fault-intolerant processor with n functional units is shown in Figure 7. Since the processor can not tolerate any faults, its model has

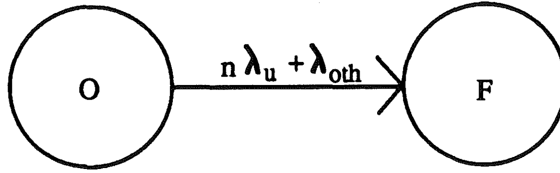


Figure 7: Markov Model of Fault-Intolerant Processor

only two states, operational and failed. The reliability of this processor is

$$R_{\text{intolerant}}(t) = e^{-\lambda_{oth} t} \cdot e^{-n\lambda_u t}$$

The general Markov model for a VLIW processor with n functional units that stores three versions of object code, one for the non-faulty situation, one for the single fault situation, and one for the double fault situation, is shown in Figure 8. Due to the fault detection and diagnosis circuitry described in the previous section and hardware support for executing the multiple versions, the failure rate of the remainder of the processor (other than the functional units) will be slightly increased relative to the fault-intolerant case. This is reflected in the modified failure rate λ'_{oth}. The labels on the states represent the number of operational functional units. F represents the failure state. There is a transition with rate λ'_{oth} from every state to the failure state since the circuitry outside the functional units represents a single point of failure. The first functional unit failure causes a transition from state n to state $n-1$. The second unit failure leaves the processor in state $n-2$. The third failure results in the entire processor failing. The reliability of this processor can be shown to be

$$R_{\text{3version}}(t) = e^{-\lambda'_{oth} t} . \left[\frac{(n-1)(n-2)}{2} e^{-n\lambda_u t} - n(n-2) e^{-(n-1)\lambda_u t} + \frac{n(n-1)}{2} e^{-(n-2)\lambda_u t} \right]$$

If we use only two object code versions thereby tolerating one fault, the reliability becomes

$$R_{\text{2version}}(t) = e^{-\lambda'_{oth} t} \cdot \left[n e^{-(n-1)\lambda_u t} - (n-1) e^{-n\lambda_u t} \right]$$

The general Markov model for a VLIW processor with n functional units that utilizes two levels of folding is shown in Figure 9. Since such a processor can tolerate any two functional unit failures, this model is quite similar to the three object code version processor model. However, note that after each functional unit failure, half

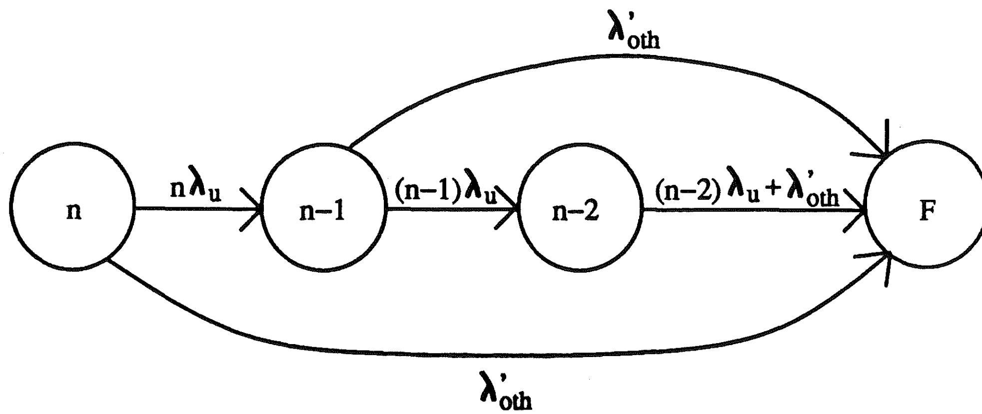


Figure 8: Markov Model of VLIW Processor using Three Object Code Versions

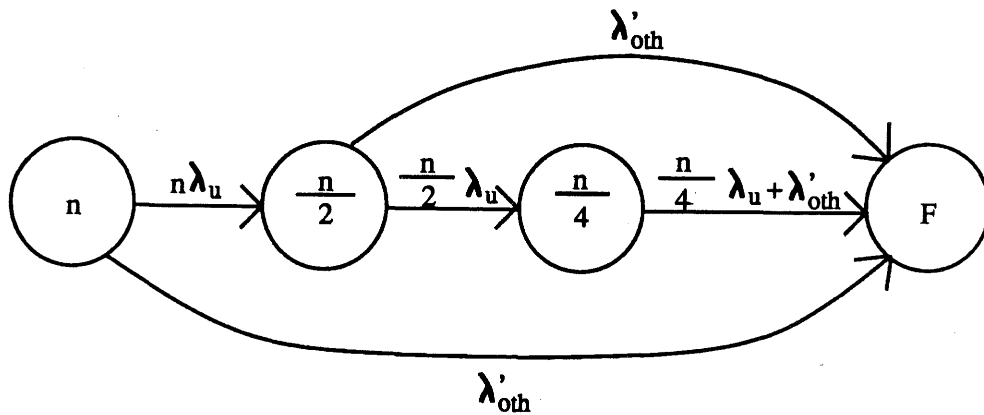


Figure 9: Markov Model of VLIW Processor with Two Levels of Folding

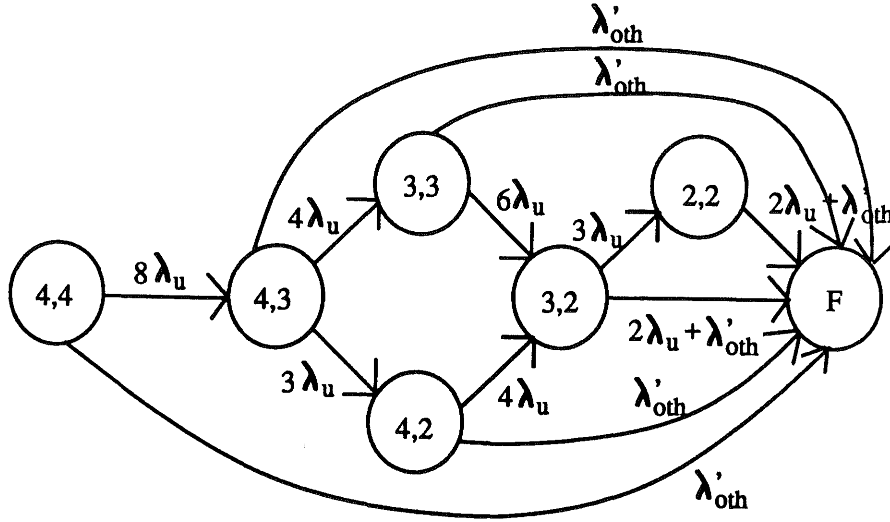


Figure 10: Markov Model of Super-Scalar Processor or VLIW Processor with Recompilation

of the units that had been operational are not used and hence the failure rate is cut in half after each fault occurrence. The reliability of this processor is given by

$$R_{\text{doublefold}}(t) = e^{-\lambda'_{oth}t} \cdot \left(\frac{1}{3}e^{-n\lambda_u t} - 2e^{-\frac{n}{2}\lambda_u t} + \frac{8}{3}e^{-\frac{n}{4}\lambda_u t}\right)$$

With only a single level of folding, the reliability is

$$R_{\text{singlefold}}(t) = e^{-\lambda'_{oth}t} \cdot \left(2e^{-\frac{n}{2}\lambda_u t} - e^{-n\lambda_u t}\right)$$

For an n-functional unit super-scalar processor or VLIW processor with recompilation, it is difficult to draw the general Markov model. The model when $n = 8$ is shown in Figure 10. The labels on the states in this model represent the number of operational functional units in each group. When this number drops below two for either group, the processor fails. Again, there are transitions with rate λ'_{oth} from every state to the failed state representing failures in the circuitry outside the functional units. This model reflects the maximum reconfiguration capability of these processors, *i.e.* up to two faulty units per comparison group can be handled. The

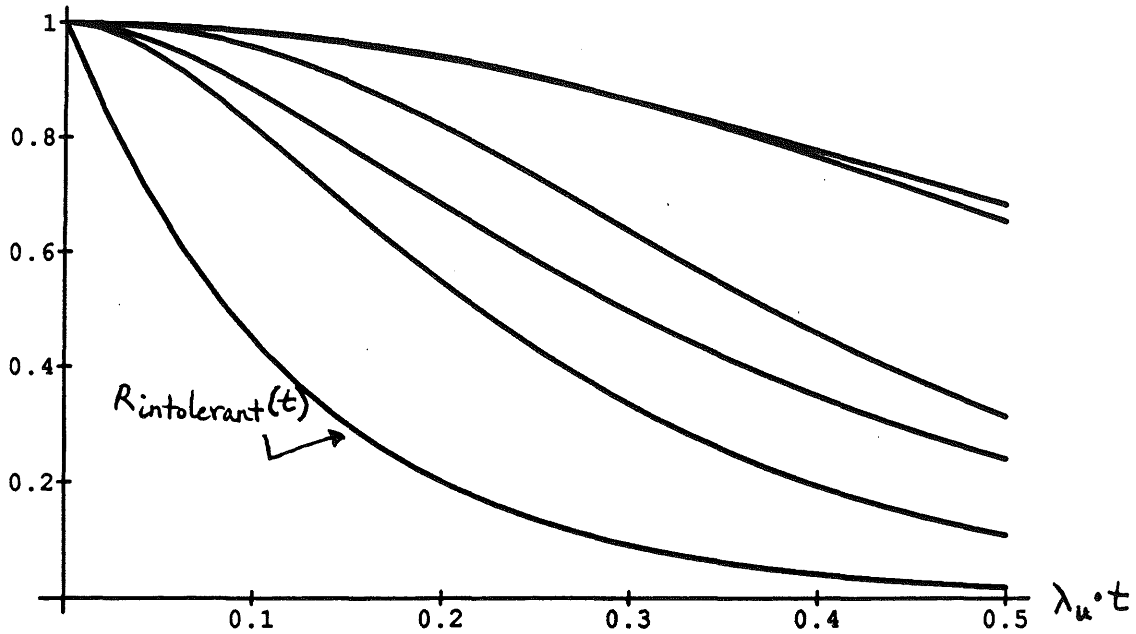


Figure 11: Comparison of Proposed Techniques with Fault-Intolerant Processor ($\lambda'_{\text{oth}} = \frac{\lambda_u}{10}$, $\lambda_{\text{oth}} = \frac{\lambda_u}{100}$, and $n = 8$)

reliability of an 8-unit processor of this type is given by

$$R_{\text{recomp/super}}(t) = e^{-\lambda'_{\text{oth}}t} \cdot \left(9e^{-8\lambda_u t} - 48e^{-7\lambda_u t} + 100e^{-6\lambda_u t} - 96e^{-5\lambda_u t} + 36e^{-4\lambda_u t}\right)$$

We have purposely omitted the derivations of these reliabilities. While some of the derivations are non-trivial, they involve well-known procedures for the transient analysis of continuous-time Markov models. The interested reader is referred to [15] for information concerning these procedures.

4.2 Evaluation

In this section, we evaluate and compare the proposed reconfiguration approaches. Figures 11–15 show plots of the six reliability functions in different ranges of time and for different choices of parameters. All plots are for eight functional units.

In Figure 11, we have chosen $\lambda'_{\text{oth}} = \frac{\lambda_u}{10}$ and $\lambda_{\text{oth}} = \frac{\lambda_u}{100}$. Thus, the failure rate of the circuitry outside the functional units is increased by a multiplicative factor of 10 by our proposed changes and the resulting failure rate is one tenth as high

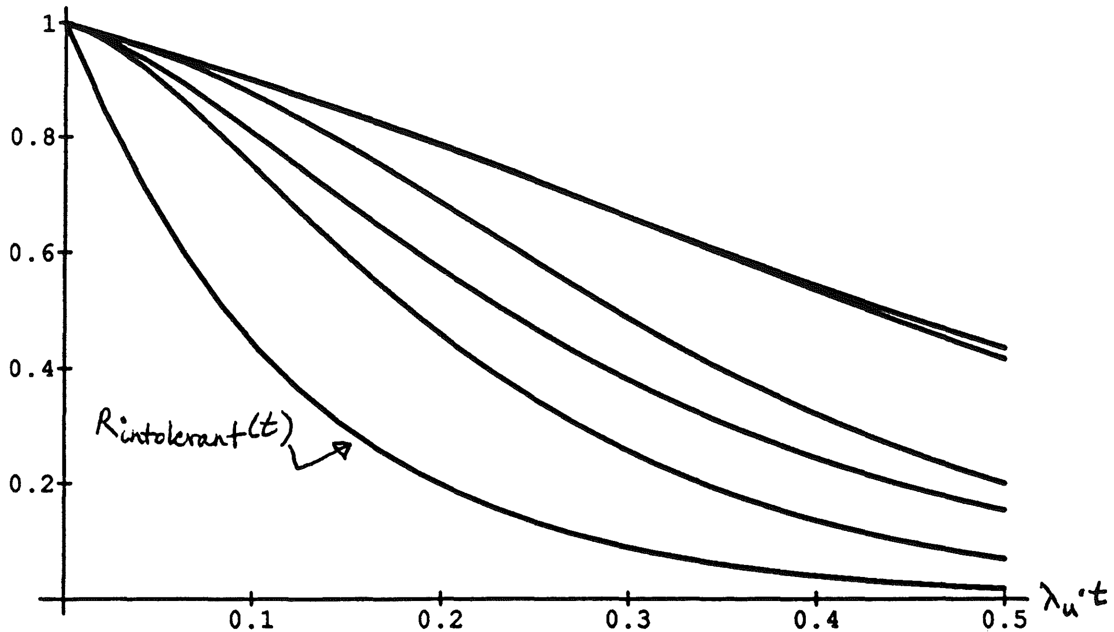


Figure 12: Comparison of Proposed Techniques with Fault-Intolerant Processor ($\lambda'_{oth} = \lambda_u$, $\lambda_{oth} = \frac{\lambda_u}{10}$, and $n = 8$)

as the functional unit failure rate. The lowest reliability curve in Figure 11 is that of $R_{intolerant}(t)$. All the proposed techniques significantly improve the processor reliability.

Figure 12 shows a similar plot with slightly different parameters. Here, $\lambda'_{oth} = \lambda_u$ and $\lambda_{oth} = \frac{\lambda_u}{10}$. Again, our proposed changes increase the "other" failure rate by a factor of ten. Now, however, this increased failure rate is identical to the functional unit failure rate. Even under these conditions, it can be seen that our proposed techniques result in significant improvement.

The range of reliability of greatest interest is usually between 0.99 and 1.0. Figures 13 and 14 show close-ups of the curves from the previous two figures in this range. The significance of the improvement of our techniques is more clearly visible in these figures. When the failure rate of the functional units is greater than the increased "other" failure rate, the worst of our techniques will survive approximately 17 times as long as the fault-intolerant processor with a probability of 0.99. In this situation, the best of our techniques lasts about 70 times as long as the fault-intolerant processor. When the increased "other" failure rate is the same

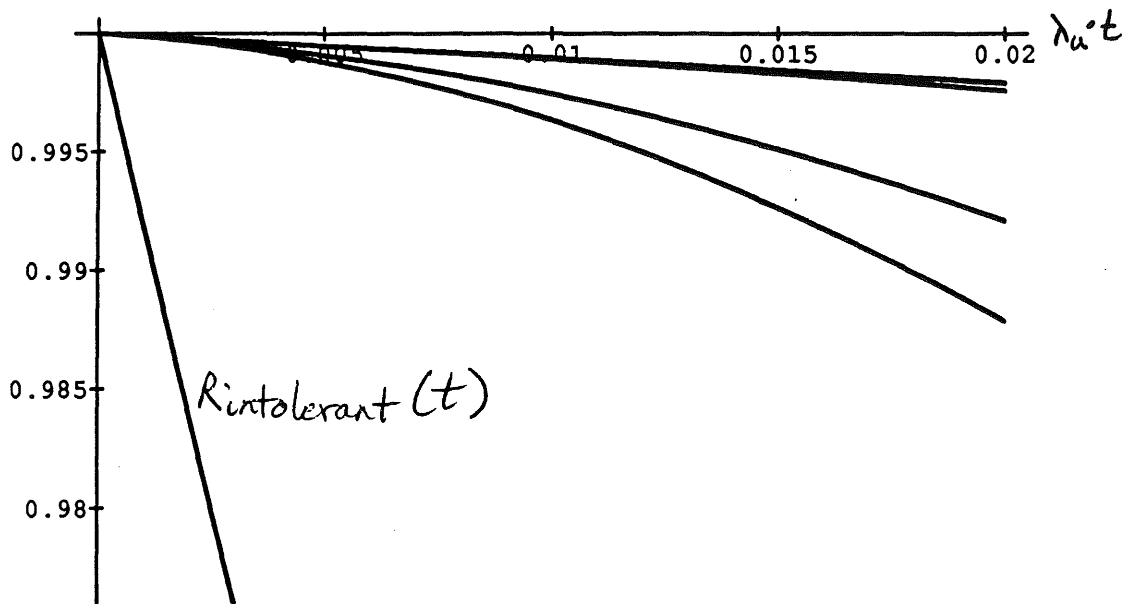


Figure 13: Comparison of Proposed Techniques with Fault-Intolerant Processor ($\lambda'_{oth} = \frac{\lambda_u}{10}$, $\lambda_{oth} = \frac{\lambda_u}{100}$, and $n = 8$)

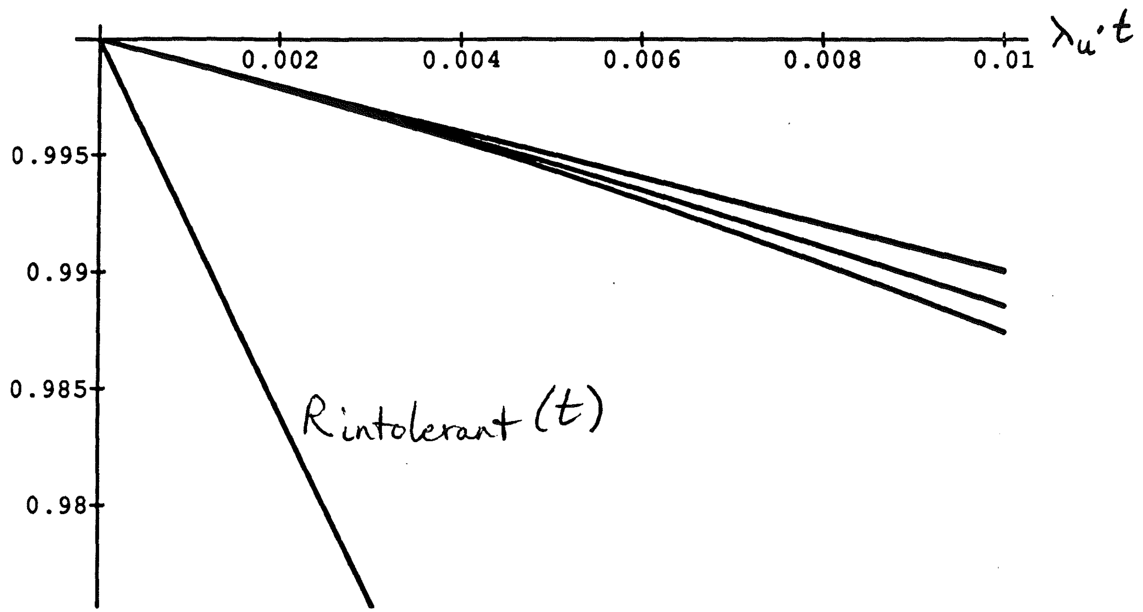


Figure 14: Comparison of Proposed Techniques with Fault-Intolerant Processor ($\lambda'_{oth} = \lambda_u$, $\lambda_{oth} = \frac{\lambda_u}{10}$, and $n = 8$)

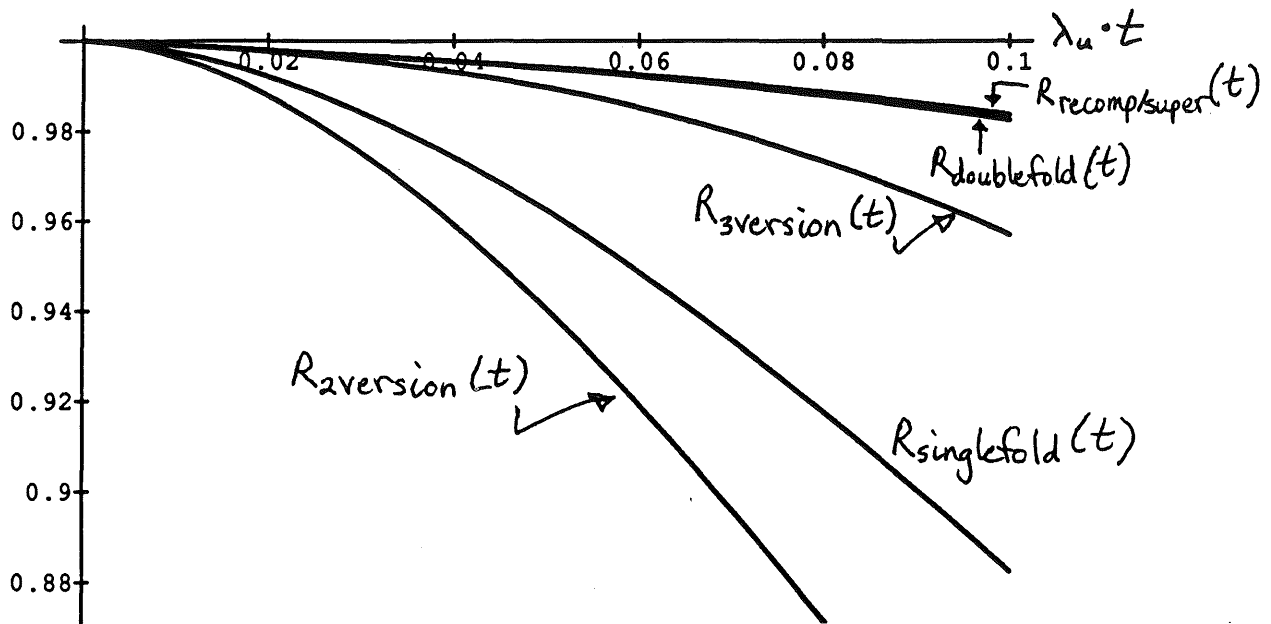


Figure 15: Comparison of Proposed Techniques
($\lambda'_{oth} = \frac{\lambda_u}{10}$, $\lambda_{oth} = \frac{\lambda_u}{100}$, and $n = 8$)

as the functional unit failure rate, the worst of our techniques lasts about eight times as long and the best technique lasts about 10 times as long as the fault-intolerant processor.

Figure 15 provides a detailed comparison of the proposed techniques. When there are two versions of object code or a single level of folding is applied, some combinations of two faulty units will cause the processor to fail. Hence, these two reliabilities are the smallest. While k versions of object code fail for any combination of $k + 1$ faulty units, the folding strategy with the same minimum level of fault tolerance can survive many combinations of $k + 1$ faulty units. Hence, from a reliability standpoint, folding appears to be superior to multiple object code versions. This is further demonstrated by the reliabilities of three object code versions and two levels of folding. Recall however that the performance degradation of folding is very severe. Hence, a choice between these two techniques would have to take into account both reliability and performance goals. When the number of functional units is eight, applying two levels of folding results in a reliability which is very nearly as good as the maximum achieved by recompilation or dynamic scheduling.

For larger numbers of functional units however, there are many fault combinations that will cause two-level folding to fail while recompilation or dynamic scheduling would not. Hence, the gap between these two approaches will widen as the number of functional units is increased.

5 Conclusion

We have presented a new approach to fault tolerance in super-scalar and VLIW processors. The approach has no performance penalty, relying solely on idle resources in the processor. Three reconfiguration strategies that trade reconfiguration time, storage overhead, and execution time, respectively, for reconfigurability were evaluated. A comparison with a fault-intolerant processor showed that significant reliability improvement can be achieved with our proposed approach. An interesting area of future research is to incorporate performance data into the reliability models of our proposed reconfiguration strategies so as to permit a performability comparison. More experimental data concerning fault coverage and fault detection latency are also needed to verify the efficacy of our approach.

Acknowledgements

The authors wish to thank Roni Potasman for providing the data in Table 1.

References

[1] P. Banerjee and J. Abraham, "Fault-Secure Algorithms for Multiprocessor Systems," *Proceedings of the International Symposium on Computer Architecture,* pp. 279–287, 1984.

[2] Chwa, K.-Y., and S. L. Hakimi, "Schemes for Fault-Tolerant Computing: A Comparison of Modularly Redundant and t-diagnosable Systems," *Information and Control,* vol. 49, pp. 212–238, 1981.

[3] R.P. Colwell, *et al.*, "A VLIW Architecture for a Trace Scheduling Compiler," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems,* 1987.

[4] A.T. Dahbura, K. Sabnani, and W. Hery, "Spare Capacity as a Means of Fault Detection and Diagnosis in Multiprocessor Systems," *IEEE Transactions on Computers,* vol. 38, pp. 881–891, June 1989.

[5] Dahbura, A. T., K. Sabnani, and L. King, "The Comparison Approach to Multiprocessor Fault Diagnosis," *IEEE Transactions on Computers,* vol. C-36, pp. 373–378, March 1987.

[6] J.-C. Fabre, *et al.*, "Saturation: Reduced Idleness for Improved Fault Tolerance," *Digest of the 18th International Symposium on Fault-Tolerant Computing,* pp. 200–205, 1988.

[7] J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers,* vol. C–30, pp. 478–490, July 1981.

[8] D. Gu, D. Rosenkrantz, and S. Ravi, "Construction and Analysis of Fault-Secure Multiprocessor Schedules," *Digest of the 21st International Symposium on Fault-Tolerant Computing,* pp. 120–127, 1991.

[9] N. Jouppi, "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and its Effect on Performance," *IEEE Transactions on Computers,* vol. 38, pp. 1645–1658, December 1989.

[10] Malek, M., "A Comparison Connection Assignment for Diagnosis of Multiprocessor Systems," *Digest of the 10th International Symposium on Fault-Tolerant Computing,* IEEE Computer Society Press, pp. 31–36, 1980.

[11] A. Nicolau, "Uniform Parallelism Exploitation in Ordinary Programs," *Proceedings of the International Conference on Parallel Processing,* pp. 614–618, 1985.

[12] A. Nicolau, "Loop Quantization: A Generalized Loop Unwinding Technique," *Journal of Parallel and Distributed Computing,* vol. 5, pp. 568–586, 1988.

[13] R. Potasman, *Percolation-Based Compiling for Evaluation of Parallelism and Hardware Design Trade-Offs,* Ph. D. Dissertation, University of California, Irvine, 1991.

[14] M. Schuette and J.P. Shen, "Exploiting Instruction-Level Resource Parallelism for Transparent, Integrated Control-Flow Monitoring," *Digest of the 21st International Symposium on Fault-Tolerant Computing,* pp. 318–325, 1991.

[15] K. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications,* Englewood Cliffs: Prentice-Hall, 1982.