HW/SW Co-Detection of Transient and Permanent Faults with Fast Recovery in Statically Scheduled Data Paths

Mario Schölzel
Department of Computer Science
Brandenburg University of Technology
Cottbus, Germany

Abstract—This paper describes a hardware-/software-based technique to make the data path of a statically scheduled superscalar processor fault tolerant. The results of concurrently executed operations can be compared with little hardware overhead in order to detect a transient or permanent fault. Furthermore, the hardware extension allows to recover from a fault within one to two clock cycles and to distinguish between transient and permanent faults. If a permanent fault was detected, this fault is masked for the rest of the program execution such that no further time is needed for recovering from that fault. The proposed extensions were implemented in the data path of a simple VLIW processor in order to prove the feasibility and to determine the hardware overhead. Finally, a reliability analysis is presented. It shows that for medium and large scaled data paths our extension provides an up to 98% better reliability than triple modular redundancy.

I. INTRODUCTION

Beside power, area and performance constrains the reliability becomes a more and more important factor for the design of embedded systems. The reason is that, due to a reduced feature size, hardware becomes more sensitive to radiation and variations during the production process. Then external events like radiation may cause transient faults, and variations in production process, metal migration and aging of the circuit may cause permanent faults. These faults may first appear in the field and not immediately after production. In order to increase the reliability of such systems, both types of faults must be handled in the field. This means that a fault must be detected, classified (transient or permanent) and, if it is permanent, the system must be aware of this fault for the rest of its lifetime. I.e., the system should avoid the usage of the faulty components or the usage of their results.

Applications that run in embedded systems very often provide a considerable amount of instruction level parallelism. For this reason, application specific processors that provide the parallel execution of operations can be used for executing such applications. This offers the opportunity to obtain high computational performance, while the clock rate and power consumption is low. In many cases, the development of such processors can be sped up significantly by using statically scheduled processors, where the compiler parallelizes the operations of the application. Very long instruction word processors (VLIW) fit into this class of processors. If application specific processors are used, then it is also possible to include techniques for detecting a transient or permanent fault as well as techniques for recovering from a fault. While a lot of work focuses on implementing application specific processors with self-repair capability on a FPGA, this paper focuses on their implementation as an ASIC. The reason is that the ASIC implementation can be much more power and performance efficient, compared with the FPGA implementation. On the other hand, the handling of permanent faults in the ASIC implementation may be more complicated. Furthermore, it is desired that the detection, localization, recovering, and masking of a permanent fault is accomplished within a few clock cycles in an embedded system, due to hard real-time demands. The contribution of this paper is a data path architecture for statically scheduled superscalar processors that is supported by software-based methods to obtain online fault detection and masking of transient and permanent faults within a few clock cycles, and with a low hardware overhead. By the combination of SW- and HW-methods, it becomes also possible for the compiler to trade performance against reliability.

II. DISCUSSION OF RELATED WORK

N-modular redundancy is probably one of the most popular techniques for detecting and masking faults in real-time systems. For N = 2 the detection of a fault (either transient or permanent) is possible. This was done for VLIW processors in the work of Holm [6] using replicated instructions. Similarly, in [1] the compiler duplicates data and instructions and inserts compare operations in order to compare the result of original and duplicated operations. Permanent faults can be detected, because the compiler schedules the original operations and their duplicates on different execution units. However, a mechanism to recover from a detected fault is not provided in [1]. The duplication of operations, data and control flow information has been also considered at source code level in [5, 11]. Because the duplication is done in the high-level-language, there is no control about the scheduling of the operations, and only transient faults can be detected. In [3] the duplication is done by a scheduler that is included in the execution stage of the pipeline...
of a statically scheduled VLIW architecture. This approach also includes a fault masking scheme but there is no control about the operations that are duplicated. In [4] the duplication of operations is done by the scheduler of a dynamically scheduled superscalar processor. Stall cycles during the execution of the program are used to re-execute already executed operations. A recovery scheme is not described, and there is no direct control about the operations which are re-executed. A better control about the duplicated operations is achieved by the approach in [7]. It uses the compiler to duplicate the original operations and to schedule them. Thus, performance can be traded against reliability. Furthermore, a special hardware extension, instead of a compare operation as in [1], is used to compare two results. However, a recovery scheme is not described. The idea, how to detect a fault, is very similar to the one described in this paper. But this paper goes one step further and provides a technique to recover from a detected fault (either transient or permanent) and masking it for the rest of the lifetime of the system, even if the fault appears online during the execution of the application. Some other work on masking a detected permanent fault in the data path of a statically scheduled processor has been published in [10], [2] and [8]. There the static schedule is modified, such that a faulty component is no longer used. In [10] and [2] this is done dynamically. In [8] an other pre-computed static schedule is used that avoids the usage of the faulty component. However, online fault detection and localization schemes are not integrated. Thus, these approaches are better used with an off-line error detection scheme.

III. FAULT TOLERANT DATA PATH ARCHITECTURE

A superscalar processor naturally provides a certain amount of redundant operators in its execution units (EUs). I.e., operators of the same type are available in more than one EU and can operate in parallel. Obviously, these redundant operators can be used to compute a result several times. If a result is computed twice on different operators, both results can be compared whether they are equal or not. Hence it is possible to detect a permanent or transient fault in one of the operators. However, in reliable systems it is not sufficient to detect a fault. It is also necessary to localize the fault, to analyze its type (permanent or transient), to provide the correct result within a short time and, if the fault is permanent, to reconfigure the system such that the faulty component is no longer used. In this section it is shown how to extend the approach in [7] such that a fault can be localized and the correct result is provided within one to two clock cycles. Moreover, it is shown how to distinguish between permanent and transient faults and how to reconfigure the system in the case of a permanent fault.

The detection of a fault is based on a static schedule of the application that ensures the double execution of all operations and contains some bookkeeping information that was generated by the compiler. I.e., every operation in the original program is duplicated by the compiler such that it is executed a second time on a different execution unit. The localization of the fault is based on a hardware extension that dynamically re-executes a duplicated operation, if there was a mismatch between the result of the original and the duplicated one. The result of the third execution is used for a majority vote.

A. Processor Model

A non-fault tolerant VLIW processor is used as reference architecture. It has a four stage pipeline (FEtch, DEcode, EXecute, Write-Back) and four homogeneous issue slots 1 to 4. Its data path is shown in figure 1. Each of the issue slots \(k\) contains two read ports (RP\(kA\), RP\(kB\)) for reading access to the register file, a fetch register (FE-Reg), a decode register (DE-Reg), an execution unit (EU), a write-back register (WB-Reg), and two multiplexer structures (BP) to implement a bypass that forwards results from the execution stage and from the write-back stage to the decode-stage. The dark gray shaded components are added to the non-fault tolerant data path in order to obtain the fault tolerant one. Their meaning is explained in section C.

![Fault tolerant data path](image)

The non-fault tolerant data path of the reference architecture is controlled by an instruction word as shown in figure 2.

![Instruction word of a VLIW with four slots](image)

In every clock cycle one instruction word is fetched from the program memory. It is composed from four parts, one for each issue slot. Each part is stored at the end of the fetch stage in the corresponding fetch register. The values of the registers \(src_{k.1}\) and \(src_{k.2}\) are stored at the end of the decode stage in the decode register together with the instruction word part from FE-Reg \(k\). The operation \(opcode_k\) is executed during the execution stage in EU \(k\), and the result of the operation is stored in the write-back register. This value is written to the destination register \(dst_k\) during the write-back stage using the write ports (WP) of the register file.

B. Fault Tolerant Instruction Word

The program that is executed by the fault tolerant VLIW processor assists the processor in the detection and localization of a fault. For this reason the compiler must generate a fault tolerant schedule from the original program by adding some bookkeeping information to each instruction word part. First, it
duplicates every operation in the original program. The scheduling algorithm of the compiler must ensure that an original operation and its duplicated operation are scheduled on different execution units. Furthermore, it must ensure that the duplicated operation is not scheduled before the original one and the result of the original operation is not used before the duplicated operation was executed. This restriction ensures that a mismatch between two results is discovered before a wrong result is used as an input. Furthermore, no difficult roll back mechanism is necessary for fault recovery. These restrictions can be modeled in the dependency graph and the resource reservation table used by the compiler during the code generation.

The fault tolerant instruction contains three additional sections in each part, which are filled by the compiler. These sections are needed by the processors hardware. An instruction word part for slot $k$ is shown in figure 3.

\[
\begin{array}{cccccccc}
\text{opcode}_{k} & \text{src}_{k,1} & \text{src}_{k,2} & \text{dst}_{k} & \text{mod}_{k} & \text{RefReg}_{k} & \text{RefEU}_{k} \\
\end{array}
\]

Figure 3. Sections of the instruction word part for issue slot $k$ with bookkeeping information.

The meaning of the sections $\text{opcode}_{k}$, $\text{src}_{k,1}$, $\text{src}_{k,2}$, $\text{dst}_{k}$ is the same as in the non-fault tolerant instruction word. These four sections are pair wise equal for an original operation and its duplicated operation. Consequently, both instructions perform the same operation with the same source operands and write to the same destination register. An example is given in figure 4. The upper black printed addition is the original operation, and the lower black printed addition is the duplicated operation. Both additions use R3 and R6 as source and R0 as destination register. The meaning of the additional sections is:

- $\text{mod}_{k}$ determines the type of the operation. If $\text{mod}_{k} = 10$, the operation is an original operation, for which a duplicated operation exists. If $\text{mod}_{k} = 11$, the operation is a duplicated one. Otherwise the operation is an original one for which no duplicated operation exists.

- $\text{RefReg}_{k}$ is a register in the temporary register file (see figure 1). It is $\text{RefReg}_{k} = \text{R7}$ for both additions in figure 4. If $\text{mod}_{k} = 10$, the result of the operation is stored into the register $\text{RefReg}_{k}$ and into the register $\text{dst}_{k}$. If $\text{mod}_{k} = 11$, the result of the operation is compared with the current value in $\text{RefReg}_{k}$. By this, a mismatch can be detected between the results of an original and duplicated operation.

- $\text{RefEU}_{k}$ is the number of the execution unit that executes the original operation if $\text{mod}_{k} = 11$. This information is necessary to re-execute the duplicated operation on an execution unit that is different from the execution units that executed the original and the duplicated operation.

C. Fault Tolerant Data Path

In order to detect a fault in an execution unit and to localize the faulty unit, the dark gray shaded components were added to the non-fault tolerant data path as shown in figure 1. These components are:

- Fault detection and compensation units (FDCL) to detect a mismatch between the results of an original and a duplicated operation and to mask an already detected permanent fault.

- A voting logic (VL) that controls the re-execution of a duplicated operation, whose result was not equal to the result of the corresponding original operation.

- A rebinding logic (RL) that maps an operation, which was fetched into slot $k$, to another slot $k'$ in order to re-execute the operation in slot $k'$.

- A temporary register file for saving temporary values. These are values from original operations whose correctness has not been verified so far. Furthermore, a valid-bit is saved with each of these values. If this bit is 1 the value is considered to be correct. Otherwise the value is considered to be wrong, i.e., it was computed by an EU for which it is already known that it contains a permanent fault.

For the rebinding logic (RL) two positions in the data path were considered. Either the rebinding logic follows the fetch register (early rebinding) or the decode register (late rebinding). The position where the rebinding takes place affects the fault recovery time as well as the reliability of the system. A discussion is given in sections D and IV. The early rebinding logic for slot $k$ is just a multiplexer structure that can select the content of any fetch register $k'$, with $1 \leq k' \leq 4$ or the opcode of the NOP-operation. The selected value is used in the decode stage of slot $k$. Thus, an instruction that was fetched to a certain slot can be remapped to any other slot, and NOP-operations can be inserted into the data path. During normal execution, the rebinding logic for slot $k$ selects the content of the fetch register $k$. Thus the instructions are executed as they were scheduled by the compiler. The late rebinding logic for slot $k$ behaves similarly. The only difference is that it also remaps the operand values, which are stored in the decode register $k$.

The voting logic (VL) contains a fault-state register for each slot $k$ where each bit corresponds to an operation that can be executed by EU $k$. This fault-bit is 1 if it is already known that the execution of the corresponding operation gives a wrong result due to a permanent fault. If the FDCL of a slot detects a mismatch, the voting logic switches the processor into the voting mode. The details about the voting mode are given in section D.

Each FDCL $k$ is located in the execution stage and gets as input the $\text{mod}_{k}$ section of the currently executed operation in slot $k$, the fault-bit $\text{fb}$ for that operation from the fault-state register $k$, the result $\text{res}$ of the operation executed in EU $k$ and the value $\text{ref}$ from the temporary register $\text{RefReg}_{k}$ together with the valid-bit $\text{vb}$ of that value. Depending on these values two general cases can be distinguished:

First, if $\text{mod}_{k} = 10$, then an original operation is currently executed, and two sub cases must be distinguished:

(a) $\text{fb} = 0$, then the result $\text{res}$ is considered to be correct and saved into the register $\text{dst}_{k}$ and into the temporary register $\text{RefReg}_{k}$ with the valid-bit set to 1.
(b) \(fb = 1\), then the result \(res\) is considered to be wrong and not saved into register \(dst_k\). But it is saved into register \(refReg\), with the valid-bit set to 0.

Second, if \(mod_k = 11\) then a duplicated operation is currently executed and four cases must be distinguished:

(a) \(fb = 0\) and \(vb = 1\), then \(res\) and \(ref\) are assumed to be correct and compared by the FDCL. If \(res = ref\) then a new fault has been detected and the voting logic activates the voting mode. If \(res = ref\), then everything is okay, and \(ref\) is saved into the register \(dst_k\).

(b) \(fb = 1\) and \(vb = 1\), then \(res\) is considered to be wrong and not saved into the destination register \(dst_k\). The register \(dst_k\) already contains the correct result that was saved by the original operation. However, if \(res = ref\), then the type of the previously detected fault in slot \(k\) was temporary and the corresponding fault-bit is set to 0.

(c) \(fb = 0\) and \(vb = 0\), then \(ref\) is considered to be wrong and the current result \(res\) is saved into the regular register file. If \(res = ref\) then the type of the previously detected fault in slot \(ref\) was temporary.

(d) \(fb = 1\) and \(vb = 0\), then at least two faults were detected in the data path. The correct result can not be determined and a global error signal is set to 1.

The cases listed above are considered for clarity in the following example for the instruction sequence in figure 4.

<table>
<thead>
<tr>
<th>+ R3</th>
<th>R6</th>
<th>R0</th>
<th>10</th>
<th>R7</th>
<th>2</th>
<th>+ R1</th>
<th>R2</th>
<th>R4</th>
<th>10</th>
<th>R0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>*</td>
<td>R4</td>
<td>R5</td>
<td>R6</td>
<td>10</td>
<td>R1</td>
<td>1</td>
</tr>
<tr>
<td>+ R4</td>
<td>R2</td>
<td>R4</td>
<td>10</td>
<td>R2</td>
<td>1</td>
<td>+ R3</td>
<td>R6</td>
<td>R0</td>
<td>11</td>
<td>R7</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4. Example of a fault-tolerant schedule.

The original operation is the addition in the first instruction with the source registers R3 and R6 and the destination register R0. It is executed on EU 1. The duplicated operation is scheduled in the third instruction with the same source and destination registers. It is executed on EU 2. Now let us assume that EU 1 and EU 2 execute the addition correctly; i.e., the corresponding fault bits are 0. The original addition writes the result into R0 and together with the valid-bit 1 into R7 (case 1a). The duplicated addition performs the same operation and compares its result with the content of R7. If there is no mismatch, then the result is written to R0 again (case 2a). Thus, R0 contains the correctly verified result. If there were a mismatch, then the voting mode would be activated, and a re-execution of the addition would start. Now let us assume that the fault-bit for the addition in EU 1 is set to 1 and the fault-bit for the addition in EU 2 is 0. When the original addition is executed, the result is not written to R0, but it is written to R7 together with the valid-bit 0 (case 1b). When the duplicated addition is executed, the FDCL finds the valid-bit in R7 set to 0. I.e., the content of R7 is not correct and therefore the result of the addition in EU 2 is written to R0 without verifying it. The permanent fault in EU 1 is masked and no delay occurs because no re-execution is done (case 2c). Now let us assume that the fault-bit for the addition in EU 1 is set to 0 and the fault-bit for the addition in EU 2 is 1. When the original addition is executed, the probably correct result is written to R0, and it is written to R7 together with the valid-bit 1 (case 1a). When the duplicated addition is executed, the FDCL 2 finds the fault-bit set to 1. I.e., the result of EU 2 is wrong. Therefore the result of the addition in EU 2 is not written to R0, and the correct result (already written by EU 1) remains in R0. The permanent fault in EU 2 is masked, and no delay occurs because no re-execution is done (case 2b).

D. Fault Localization

This section outlines the behavior of the voting mode. This mode is activated by the voting logic in order to localize a faulty unit after a mismatch has been detected by the FDCL in clock cycle \(t\). It is assumed that the mismatch is detected during the execution of the duplicated operation \(v\) on EU \(k\). The corresponding original operation \(v'\) was executed on EU \(k'\). Furthermore, the late rebinding is implemented. The processor switches immediately after detecting the mismatch into the voting mode. I.e. the fetch and decode registers are stalled in clock cycle \(t+1\) and keep their values. The voting logic determines an execution unit \(r\) with \(r \neq k\) and \(r \neq k'\) that can execute operation \(v\) (i.e. the fault-bit of operation \(v\) is not set to 1 in the fault-state register of slot \(r\) ). The voting logic sets the control signals of the rebinding logic \(r\) such that in slot \(r\) operation \(v\) is re-executed in clock cycle \(t+1\). The control signals of all other rebinding logics are set such that the corresponding slots execute a NOP-operation in clock cycle \(t+1\). Please note the rebinding logic also maps the \(mod, refEU\) and \(refReg\)-section of operation \(v\) to slot \(r\). Thus the result of EU \(r\) is compared again with the value of \(refReg\). If the FDCL \(r\) does not detect a mismatch during the re-execution, then the value in \(refReg\) is correct and EU \(k\) computed a wrong value. Therefore, the fault-bit for operation \(v\) in the fault register \(k\) is set to 1. If the FDCL detects a mismatch during the re-execution, then the value in \(refReg\) was not equal to the result of EU \(k\) and not equal to the result of EU \(r\). Because the probability that all three results are different is very low, the result of EU \(k'\) is considered to be wrong, and the fault bit of operation \(v\) in the fault register \(k'\) is set to 1. In clock cycle \(t+2\) the processor switches back to the normal operation mode and executes the operations stalled in the fetch-register. Therefore, the re-execution takes only one clock cycle.

If the early rebinding is used, the re-execution takes two clock cycles. This is because after re-binding the operands of operation \(v\) must be loaded again from the register file during the decode stage. The fetch register must contain a shadow fetch register to roll back the execution by one instruction.

IV. RESULTS

The fault tolerant VLIW architecture has been implemented in VHDL, and several fault conditions were simulated successfully to prove the feasibility. The reliability of five VLIW systems have been compared. These systems are the fault tolerant VLIW system with early re-binding (FTVLIW_early), with late re-binding (FTVLIW_late), a non-fault tolerant VLIW system with four slots (VLIW4), with two slots (VLIW2), and a TMR-system (TMR) based up on the non-fault tolerant VLIW system.
with two slots. The computational performance of the VLIW2 and the TMR is equal to both FTVLIW systems. The TMR system and the VLIW2 can execute two original operations per clock cycle. The same is true for both FTVLIW systems, if each operation is duplicated and scheduled at the same instruction as the original ones. The VHDL models of all five systems were synthesized using a 250 nm CMOS library. The synthesis results are shown in table 1 for the VLIW4, VLIW2, FTVLIW early and the FTVLIW late systems.

TABLE I. TRANSISTOR COUNTS OF FOUR VLIW SYSTEMS WITH SMALL SIZED EUS.

<table>
<thead>
<tr>
<th>Component</th>
<th>VLIW4</th>
<th>FTVLIW late</th>
<th>FTVLIW early</th>
<th>VLIW2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL</td>
<td>800</td>
<td>1</td>
<td>800</td>
<td>1</td>
</tr>
<tr>
<td>FE-Reg</td>
<td>2300</td>
<td>1</td>
<td>2300</td>
<td>1</td>
</tr>
<tr>
<td>DE-Reg</td>
<td>1300</td>
<td>4</td>
<td>2200</td>
<td>4</td>
</tr>
<tr>
<td>BP</td>
<td>1400</td>
<td>8</td>
<td>1400</td>
<td>8</td>
</tr>
<tr>
<td>EU</td>
<td>4000</td>
<td>4</td>
<td>4000</td>
<td>4</td>
</tr>
<tr>
<td>WB-Reg</td>
<td>500</td>
<td>4</td>
<td>500</td>
<td>4</td>
</tr>
<tr>
<td>BP</td>
<td>6900</td>
<td>8</td>
<td>6900</td>
<td>8</td>
</tr>
<tr>
<td>WP</td>
<td>34800</td>
<td>1</td>
<td>34800</td>
<td>1</td>
</tr>
<tr>
<td>RL</td>
<td>-</td>
<td>1500</td>
<td>900</td>
<td>-</td>
</tr>
<tr>
<td>VL</td>
<td>-</td>
<td>3200</td>
<td>3800</td>
<td>-</td>
</tr>
<tr>
<td>TRP</td>
<td>-</td>
<td>1900</td>
<td>1900</td>
<td>-</td>
</tr>
<tr>
<td>TWP</td>
<td>-</td>
<td>8000</td>
<td>8000</td>
<td>-</td>
</tr>
<tr>
<td>FDCL</td>
<td>-</td>
<td>600+100</td>
<td>600+100</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>111500+EUs</td>
<td>142700+EUs</td>
<td>134300+EUs</td>
<td>54500+EUs</td>
</tr>
</tbody>
</table>

Column T is the transistor count of the corresponding component rounded to the nearest multiple of hundred, and column # is the number of instances of that component in the corresponding VLIW system. It is assumed that the memory elements in the register files are protected by other techniques, e.g. error correction codes. Therefore, their transistor count is not included in table 1. All components, except CTRL, can be found in figure 1. CTRL is the control path of the VLIW system.

The reliability $R_T$ of a single transistor $T$ at time $t$ is $R_T = e^{-\lambda t}$ if the failure rate $\lambda$ is constant [9]. The reliability of the VLIW systems is computed based on $R_T$. For this reason the components of a VLIW system, which are composed of transistors, are partitioned into two types; the non-fault tolerant (not bold in table 1) and the fault tolerant (bold in table 1). If one of the non-fault tolerant components fails, the whole system will fail. The number of transistors in the non-fault tolerant components is denoted by $T_{nft}$. Each fault-tolerant component belongs to a certain slot. Its functionality can be overaken from a component in another slot. Thus, all fault tolerant components of one slot may fail, but the fault tolerant VLIW system will not. The total number of transistors in the fault-tolerant components of a slot is denoted by $T_{ft}$. Given $T_{nft}$ and $T_{ft}$, the reliability $R_T$ of a VLIW system $S$ with $n$ slots at time $t$ is computed by

$$R_T = R_T^{T_{nft} + \times T_{ft}} + R_T^{T_{nft} \times n} \cdot R_T^{(n-1)T_{ft}} \cdot (1 - R_T^{T_{ft}}).$$

The values for $T_{nft}$, $T_{ft}$ and $n$ are given in table 2 for the five VLIW systems. In the following the number of transistors in component $c$ is denoted by $T_c$. Obviously $T_{ft} = 0$ for the non-fault tolerant systems VLIW2 and VLIW4. The TMR system is obtained from the VLIW2 by tripling both slots of the VLIW2. I.e., each slot in the TMR system is composed of two read ports, one decode register, one EU, one WB-Reg and two bypasses. Thus $T_{ft} = 2 \cdot T_{BP} + T_{DE-Reg} + T_{EU} + T_{WR} + 2 \cdot T_{BP} = 17000 + T_{EU}$.

Any fault in slot $k$ that occurs in the EU, the re-binding unit or the read port of the temporary register file of the FTVLIW late will be detected. Thus $T_{ft} = T_{EU} + T_{WR} + T_{BP} + T_{FDCL} = 14000$. Please note, a fault that occurs in the bypass (BP) or read ports (RP) of the regular register file of slot $k$ will be detected but may not be localized correctly. This is because such a fault may corrupt the data, which are stored in the decode register of slot $k$. A re-execution of the duplicated operation will use these corrupted data again and give the wrong result again, which is then selected as the correct one by the majority vote. Thus, $T_{nft} = 126700$. If the early rebinding is used, the re-execution of an operation causes the loading of the operands from the register file through other read ports, because the decode stage is repeated for the re-executed operation. For this reason, faults in the read ports and the bypass of a slot will be detected and localized correctly. Thus for the FTVLIW early system is

$$T_{ft} = T_{EU} + T_{WR} + 2 \cdot T_{BP} + T_{FDCL} + T_{DE-Reg} + 2 \cdot T_{BP} + T_{TRP} = 22200 + T_{EU}.$$ 

The number of non-fault tolerant transistors in slot $k$ is 600, and therefore $T_{nft} = 54600$. The reliability of these four systems is shown in figure 5 for small, medium and large sized execution units. A small sized EU consists of about 4000 transistors and implements only logical operations, integer addition and subtraction. A medium sized EU of about 20000 transistors will support more complex integer operations, e.g. a multiplication. A large sized EU of about 100000 transistors supports complex floating point operations.

The reliability analysis shows that the early rebinding is always better than the late rebinding. For small execution units, the TMR system has always a much better reliability than the FTVLIW early. For small fault rates ($\lambda < 20 \cdot 10^{-7}$) the TMR and FTVLIW early system have a slightly better reliability than the non-fault tolerant VLIW2. This situation changes for medium sized EUs. The TMR system still has a better reliability than the FTVLIW early. But for $\lambda < 58 \cdot 10^{-7}$, the reliability of the FTVLIW early is up to 4.8% better than the reliability of the VLIW2 system. For large scaled EUs, the FTVLIW early outper-
forms the TMR approach. For \( \lambda < 3.5 \times 10^{-7} \), the TMR system still has a slightly better reliability than the FTVLIW early (only about 0.3%). But for \( \lambda \geq 3.5 \times 10^{-7} \) the FTVLIW early has a much better reliability, which is up to 98% higher than the reliability of the TMR system at that point where the reliability of the VLIW early is equal to the reliability of the VLIW2. For \( \lambda > 53 \times 10^{-7} \), the reliability of the TMR and FTVLIW early becomes smaller than the reliability of the VLIW2.

It can be concluded that for small sized EUs the TMR approach is the best one. For medium sized EUs, the reliability of the VLIW early is close to the reliability of the TMR approach. However, it provides much more flexibility. I.e., the compiler can trade performance against reliability, because not every operation must be duplicated. For large sized EUs, the FTVLIW early approach is the best one for fault rates up to \( 5.3 \times 10^{-6} \). For higher fault rates, the reliability of all fault tolerant systems is smaller than the reliability of a non-fault tolerant system.

V. CONCLUSION

A technique to improve the reliability of statically scheduled superscalar processors with medium to large scaled execution units was presented. Transient as well as permanent faults can be detected online. A fault recovery is possible within one to two clock cycles. Detected permanent faults can be masked in order to avoid a repeated fault recovery. Thus, the presented approach is well suited for the design of reliable real-time systems. Moreover, the architecture provides the opportunity that the compiler can trade performance against reliability by selecting only certain operations for duplication. Thus some parts of an application may run with high reliability while other parts benefit from higher performance by using all 4 EUs.

VI. REFERENCES