A Delay Estimation of Rescheduling Schemes for Static Scheduled Processor Architectures

M. Schölzel, Brandenburg University of Technology, Computer Engineering Group, Cottbus, Germany,
E-Mail: mas@informatik.tu-cottbus.de

Abstract

We compare three different rescheduling schemes for statically scheduled processor architectures. One of the rescheduling schemes is software-based while the others are based on hardware support. The rescheduling becomes necessary if the compiler generated schedule for a static scheduled processor architecture must be changed in-the-field because of a permanent fault in the data path of the processor. By comparing the hardware and software-based rescheduling schemes we can show that our proposed software-based rescheduling scheme in many cases reduces the worst-case latency of the executed program in the presence of a permanent fault.

1 Introduction

The development of silicon IC technology during the last decades has yielded an unprecedented exponential improvement of the performance per cost ratio for integrated circuits and devices. Complex integrated systems on a chip (SoCs) have become the mainstay of electronics in many applications, specifically automotives, appliances, and communications. Recent forecasts for the properties of circuits that have a feature size below 50 nm predict several critical features [2, 5, 7, 21, 29, 30]:

- First, relative deviations of device- and circuit parameters such as transistor threshold voltages will increase due to quantum mechanical effects, making a certain share of basic devices non-functional even in the absence of physical defects [5, 29]. Such effects will result in permanent faults of devices.
- Second, nano-scale devices are going to exhibit a stronger vulnerability to distorting influences such as radio-active particle radiation from cosmic or terrestrial sources [3, 8]. Such effects are likely to produce transient fault and error conditions without a permanent damage.
- Third, nano-scale devices are likely to have a higher level of inherent stress conditions due to higher electric field strength in insulation layers and higher current density on interconnects. This may result in permanent faults that occur in the field.

As a consequence, devices which are fully operational after production have a higher probability of failure by a later point of time. Therefore nano-electronic systems that are used in long-living and safety-critical applications are likely to need architectures that can repair permanent faults in the field by a kind of build-in self-repair (BISR) capability [4, 11, 22, 28, 29]. But also the increasing complexity of SoCs makes BISR capability a must for future designs. The semiconductors industry's ITRS Roadmap predicts that, due to the high complexity of SoCs, a full functional test after production becomes extremely expensive or even impossible [1]. Production costs could be reduced by relaxing the requirement for 100% correct devices. Then architectures are needed that facilitate repair functions at least after production, and, considering both test escapes and wear-out in operation, built-in test and self-repair in the field.

Furthermore, in recent years application specific processors became more and more popular as components in SoCs, due to their good ratio between power consumption, performance and flexibility. Due to the required self-repair functionality of SoCs, also a self-repair possibility for the included processors is necessary. The simplest solution would be to replace the whole processor by a backup processor in the case of a permanent fault. However, this would be also the most expensive solution. For this reason, we consider application specific statically scheduled programmable processor architectures with instruction level parallelism (e.g. a very long instruction word processors - VLIWs), and how to make them tolerate permanent faults that may occur in their data path in the field. Usually this type of architecture contains many redundant operators in the data path. Thus, if there occurs a permanent fault in an operator, it may not be necessary to replace the whole processor. Rather another operator may overtake the work of a faulty one. A picture of our supposed architectural framework is shown in figure 1.

It uses a 4-stage instruction pipeline with the well known stages *fetch*, *decode*, *execute* and *write-back* and supports instruction level parallelism. I.e., the data path contains a single register file that is accessed by several function units (FUs) that run in parallel. Each of the FUs contains several operators (e.g. adder, multiplier) and can execute one operation (e.g. addition, multiplication) per clock cycle.

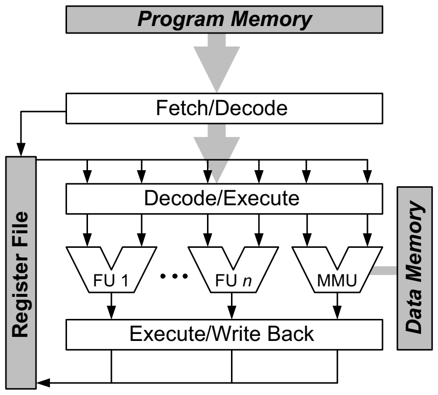

Figure 1: Data Path of our supposed static scheduled processor architecture.

The operations are scheduled statically (i.e. at compile time) by the compiler into instructions, and each operation is also bound statically to a FU. Thus, in each clock cycle, one instruction is executed by the processor as it was generated by the compiler. A schematic picture of an instruction is shown in figure 2.

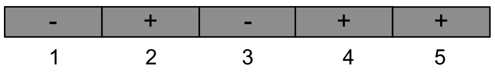

Figure 2: Example of an instruction that contains five operations. The number below each operation is the number of the FU to which the operation is bound.

The advantage of statically scheduled processor architectures in the context of application specific processor design is its simple control path that can be generated automatically. The control path can be kept very simple, because all work according to scheduling, binding and hazard-avoidance, is done by the compiler. The disadvantage of statically scheduled processor architectures in the context of fault-tolerant computing is that an operation, that is bound to a faulty FU, can not simply be executed by another FU.

2 Related Work

Fault-tolerant computing is an established research area, and a lot of techniques for detection and handling of transient faults have been developed and implemented [15, 16, 20]. These techniques may be also adequate for detecting a permanent fault, but not for repairing it permanently. For this reason, a faulty component in a self-repairing system must be identified and its allocated tasks must be moved to other components. This requires either a reconfiguration of the interconnects in the data path or a redistribution of tasks, i.e. in our architectural framework a new binding[1] in the static schedule in order to avoid the usage of the faulty component.

Hardware can be reconfigured in the field by providing mechanisms for activating and deactivating certain parts of the hardware (e.g. by pass-transistor or fuses) together with mechanisms for re-routing the data [18, 19, 25]. Such approaches require a sophisticated configuration data management and a considerable hardware overhead for configuring the system by selectors, pass-transistors or transmission gates. They can be integrated directly into the design of a special ASIC, but they require either a relatively regular structure of an ASIC, or they produce a high hardware overhead for control structures.

Another popular technique is provided by field-programmable gate arrays (FPGAs). The usage of a faulty area in such devices can be avoided by moving functionality from a faulty area to a non-faulty area by a reconfiguration that is performed by a repair procedure [23]. This reconfiguration can be done in the field. However, in order to have enough area available for a reconfigured system, the original FPGA must provide backup areas. Thus, a lot of resources are unused and preserved for the repair capability [12, 23]. Furthermore, the area and power consumption overhead for implementing a system in a FPGA is about an order of magnitude compared to the implementation in optimized digital hardware.

Another possibility is simply to avoid the usage of a faulty component in the data path instead of reconfiguring the data path. A processor can avoid the usage of a faulty component by rebinding and/or rescheduling operations in the compiler generated schedule. This means that the compiler generated schedule must be changed. This is done in existing approaches by rebinding and rescheduling[2] the operations in the schedule either in hardware [6] or by pre-computing several schedules, one for each possible fault situation [13, 14].

In the first case, a rebinding or even a rescheduling and rebinding is done on-the-fly in hardware before each instruction is executed. This requires a substantial amount of additional hardware in the control path of the processor and leads to the loss of one of the big advantages of a static scheduled architecture; the simplicity of its control path. As mentioned in section 1, this simplicity makes statically scheduled processor architectures very attractive as an architectural framework for scalable application specific processors that are adapted to a given set of applications by a design space exploration. Thus, the simplicity of the control path should be maintained.

In the second case, the usage of a faulty component is avoided without hardware support. I.e., another schedule, which was pre-computed, is executed. This schedule does not use the faulty component. Such an approach has two important drawbacks. First – depending on the application – maybe many schedules must be pre-computed and saved in the program memory. Second, these schedules must be available for the whole application. This becomes impractical especially for large applications.

[1] Changing the binding of an operation to a functional unit means that the operation is executed in the same instruction but by another FU.

[2] Changing the time in the schedule (i.e. the instruction) when the operation is executed.

3 Software-Based Rescheduling

To overcome the mentioned drawbacks, we propose a new software-based rescheduling scheme that can be executed in-the-field. It rebinds the operations in each instruction if a permanent fault is detected in the data path of the processor. In this paper we present an evaluation which shows that our approach decreases the worst-case delay during the execution of the program, compared to the hardware based rescheduling schemes, but still keeps the architectural framework simple. Moreover, only a single schedule for the application must be generated by the compiler. This schedule is adapted in-the-field by the software-based rescheduling scheme to an occurred fault situation. Therefore, pre-computation of many schedules is not necessary.

The remainder of this paper is organized as follows. In the next sections we introduce the three rescheduling schemes that we use for our estimation. Then we give an overview about the overall system architecture that implements the software-based rescheduling scheme. Finally, we explain our experimental setup, give results for some benchmarks and draw our conclusions.

3.1 Rescheduling Schemes

We have evaluated three different rescheduling schemes. Two of them are hardware based. We use them for comparison with our software-based rescheduling scheme.

Simple Hardware Solution (SHS)

The SHS extends the decode stage of the processor pipeline such that, in the case of a permanent fault, the following behaviour is obtained: We assume a detected permanent fault in an operator of type x in functional unit a. If the current instruction in the decode stage requires the execution of an operation of type x on FU a, that operation is delayed by one clock cycle. All other operations of the current instruction are routed to the execution stage, and the next instruction is delayed for one clock cycle. Thus, in the next clock cycle, an empty instruction can be routed to the execution stage by the control logic that contains only the delayed operation. The delayed operation is executed on a FU that contains a working operator of type x. This rebinding is computed by the control logic of the processor.

The behaviour is illustrated in the example in figure 3 where we have a data path with five functional units (shown in the execution stage) including the shown operators. Furthermore, the operator for the multiplication in FU 5 is assumed to be faulty. The instruction at the beginning of the execution stage in figure 3 is executed as it was fetched, because it does not use the faulty operator in FU 5. However, the instruction at the beginning of the decode stage would use the multiplier in FU 5. Therefore the control logic delays the multiplication, and the instruction at the beginning of the fetch stage.

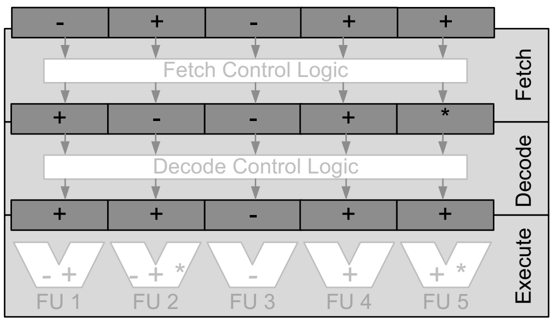

Figure 3: Example for the behavior of the simple hardware rescheduling scheme.

Figure 4 shows the configuration of the data path in the next clock cycle.

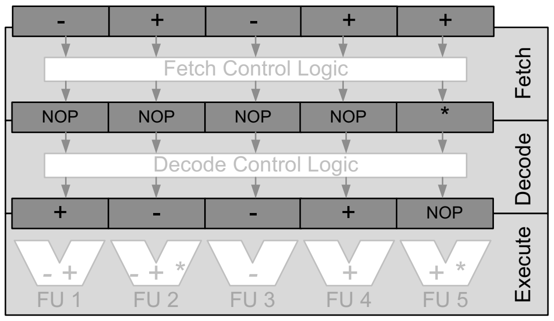

Figure 4: Example for a delayed multiplication.

The multiplication has been delayed, and all other operations were routed to their corresponding FU. FU 5 executes a NOP (no operation) instead of a multiplication. The multiplication is executed one clock cycle later as shown in figure 5.

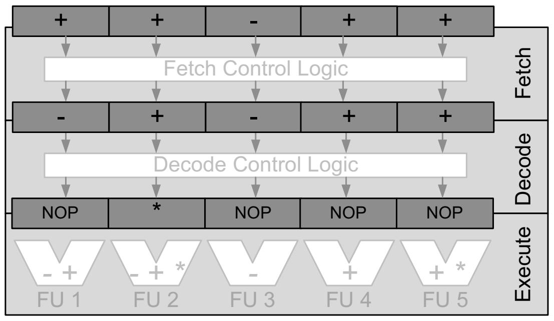

Figure 5: Example for executing the delayed multiplication.

It must be noticed that the source operands of the multiplication must be read from the register file by the same time while the operands for the non-delayed operations are read. This is necessary to avoid overwriting of these operands in the register file with results from the execution stage.

Complex Hardware Solution (CHS)

The complex hardware rescheduling scheme is an extension of the simple one presented in the previous section. The decode logic is modified such that, in the case of a detected permanent fault in an operator of type x in functional unit a, the following behaviour is obtained: If the current instruction in the decode stage requires the execution of an operation of type x on a FU a, and there is another FU b with operator x that executes a NOP in that instruction, then the operation of type x is executed on FU b. Therefore, no delay will occur.

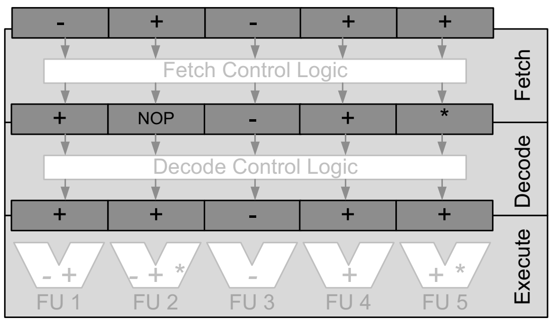

Figure 6: Example for the behavior of the complex hardware rescheduling scheme.

For example, the instruction at the beginning of the decode stage in figure 6 can not be executed as it was scheduled by the compiler, because the multiplication would be executed on the faulty multiplier in FU 5. However, the multiplication can be re-routed to FU 2 and be executed there. Thus, no delay will occur. Figure 7 shows the modified instruction as it is routed to the execution stage in the next clock cycle.

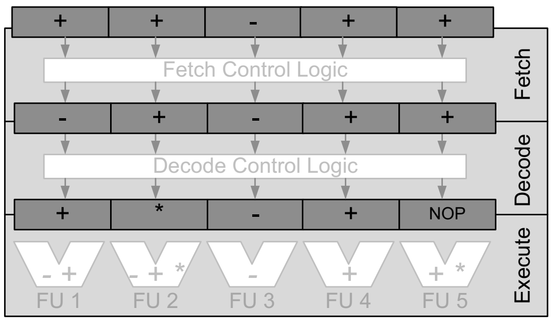

Figure 7: Example of re-bounded multiply-operation.

However, if there is no such FU b available, the operation of type x must be delayed as in the simple hardware solution. An example for such a situation is given in figure 3. Thus the complex hardware solution is an extension of the simple hardware solution. Rebinding and delayed execution is controlled by the control logic of the processor on-the-fly.

Software Solution (SWS)

The software rescheduling scheme allows a more complex rescheduling, because it is not done by hardware, but by a software routine that is executed on a processor[3]. The software routine computes a rebinding of the operations for each instruction of the program. Again, we assume that there is a fault in the operator of type x in FU a. Then the rebinding is a permutation of the operations in each instruction, such that FU a with the faulty operator x does not execute an operation of type x. For example, the instruction in figure 8 (b) can not be executed, if the multiplier in FU 5 in figure 8 (a) is faulty. But the instruction in figure 8 (c) with permutated operations can be executed.

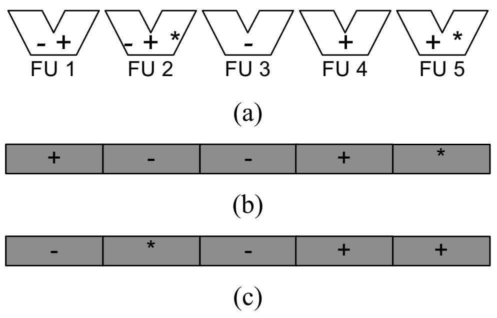

Figure 8: (a) Configuration of the data path. (b) Original Instruction. (c) Instruction with permutated operations.

As for the complex hardware solution, the statically scheduled processor must provide a mechanism as in the simple hardware solution, because such a permutation may not exist. For example, consider the instruction in figure 9.

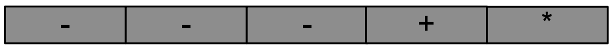

Figure 9: Example of a not permutable instruction.

Again we assume the multiplier in FU 5 in figure 8 (a) to be faulty. Then there exists no permutation of the operations such that the multiplication can be executed by FU 2, because the FUs 1 to 3 must execute a subtraction.

In contrast to the hardware solutions, the rebinding is computed for the binary program, which is saved in the program memory. I.e., the processor must have read and write access to its program memory, and the permutation cannot be computed on-the-fly.

3.2 Software Rescheduling Algorithm

The permutation can be computed efficiently. The algorithm is presented in the following. We can model the problem of finding a permutation for a given instruction w and an occurred fault in FU f as a rebinding-graph. A rebinding-graph is a directed graph (N,E), where N is a set of nodes and $E \subseteq N \times N$ is a set of edges. The nodes represent the functional units in the data path. An edge (u,v) represents the possibility that an operation, which is executed on FU u, can be executed on FU v, too. Thus, there is an edge (u,v) from node u to node v, if and only if:

- FU u executes an operation of type t in instruction w and
- FU v includes an operator of type t and
- the operator of type t in FU v is not faulty and
- $u \neq v$.

[3] For the moment let us assume that there is such a processor. Which processor can be used for this task is discussed in section 3.3.

In order to compute the permutation, let G be a set of functional units that contains the FU f (i.e. the functional unit which is faulty) and all FUs that execute a NOP in the given instruction w.

The goal is to find a path in the constructed rebinding-graph, with the source node f and a sink node that is in the set G. The permutation is obtained by shifting each operation that is executed by an FU on that path along one edge on that path to another FU. By the construction of the graph we made sure that the operation can be executed on the FU to which it was shifted. Furthermore, the last node on the path is either a FU that executed a NOP or the first node of the path from which we have shifted away the executed operation. Consider the following example. We use the data path from figure 8 (a) and the instruction from figure 8 (b) and we assume that the multiplier in FU 5 is faulty. The corresponding rebinding-graph is shown in figure 10.

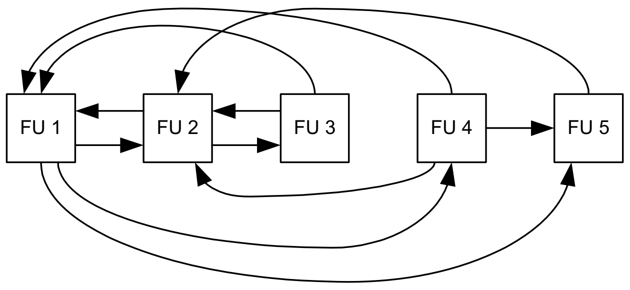

Figure 10: Example of a rebinding-graph.

For example, the multiplication that is executed on FU 5 can also be executed on FU 2. Thus, we have the edge (5,2). The subtraction that is executed on FU 2 can also be executed on FU 1 and FU 3. Hence, we have the edges (2,1) and (2,3). In this example is $G = \{5\}$, because the instruction in figure 8 (b) contains no NOP. A path that starts at source node 5 and ends at a node in G is 5, 2, 1, 5. Now we have to shift each operation from the instruction in figure 8 (b) along one edge on the path 5, 2, 1, 5. I.e., operation * is shifted from FU 5 to FU 2, operation – is shifted from FU 2 to FU 1 and operation + is shifted from FU 1 to FU 5. By doing this we obtain the permutation that is already shown in figure 8 (c).

In order to find the requested path in a given rebinding-graph for a given source node s and a given set G, formally we have to compute the transitive closure E^+ of the edges E and check whether $(s,d) \in E^+$ for any $d \in G$ or not. The solution can be obtained faster, if a breadth-first-search (bfs) in G is performed starting at node s. For all nodes u that are reached by the bfs, we determine the next unreached nodes v that can be reached from a node u via an edge and keeping in mind for each of these nodes v the node u from which v was reached. The bfs ends, if we reach a node that belongs to G, or we can not reach a new node. In our example, the bfs would start at node 5. From node 5 we can reach only node 2, keeping in mind that we arrived from node 5. From node 2 we can reach nodes 1 and 3, keeping in mind that we arrived from node 2. From node 3 we can not reach a new node (i.e., a node we have not reached before). But from node 1 we can reach nodes 4 and 5, keeping node 1 in mind. The bfs ends because node 5 belongs to G. With the nodes we kept in mind we can reconstruct the path in the reverse order. This means we go back from node 5 to node 1, then from node 1 to node 2, and finally from node 2 to node 5.

3.3 Overall System Architecture

Up to now we have explained how the software rescheduling algorithm should work, but not on which processor it should be executed. Two possible system architectures will be discussed in this section. The first architecture is shown in figure 11.

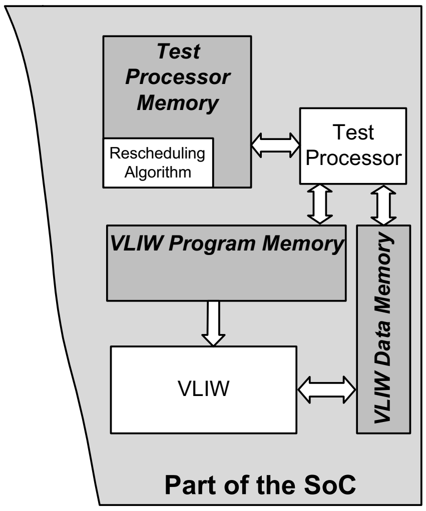

Figure 11: System Architecture where the SoC contains an additional test processor.

The SoC contains an extra test processor [9, 10]. This is a very simple and small processor[4] that is used in the field for executing and controlling a self-test-routine in the SoC. This test processor can also be used for testing the VLIW processor. This can be done by initiating a test routine in the VLIW that computes several values in its data memory. From there, the values can be read and compared with the values computed by the test processor. If a fault is detected, the test processor executes the software rescheduling algorithm. However, it needs access to the program memory of the VLIW processor in order to modify the instructions in the program memory of the VLIW. This type of architecture does not need extensive modifications in the VLIW itself. The most expensive modification is the read- and write-access to program and data memory of the VLIW processor.

The second system architecture is somewhat more unusual and shown in figure 12. There, the VLIW-processor itself executes the software rescheduling algorithm. But how can

[4] Please note, that the computation performance of the VLIW is much bigger than that of the test processor. Thus, the test processor is not able to overtake the work of the VLIW.

a processor with a faulty component execute the rescheduling algorithm in a correct manner? Please recall that the VLIW supports a simple hardware rescheduling scheme. This means, even in the presence of a permanent fault in the data path, the rescheduling algorithm is executed in the right way, because the hardware rebinds operations of that algorithm to other FUs on-the-fly if this is necessary.

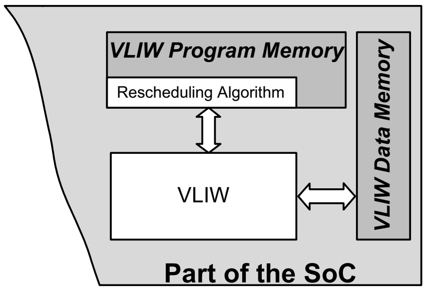

Figure 12: System Architecture where the VLIW processor itself executes the software rescheduling algorithm.

However, this may increase the execution time of the rescheduling algorithm. One drawback of this system architecture is that more modifications in the control path of the VLIW are necessary. I.e., the VLIW itself needs write-access to its own program memory, which is not common in Harvard-Architectures. This can be accomplished by two new machine instructions. The first one loads an instruction word in a consecutive sequence of processor registers. The rescheduling algorithm can perform the needed modifications in the instruction word and by a second new instruction the same or modified instruction word can be written back to the program memory. The advantages of this system architecture are:

- No extra test processor is needed.
- The full performance (except for the faulty FU) of the VLIW processor is used for rescheduling.

The detection of permanent faults in the second system architecture can be done by a software-based self-test [17].

4 Experimental Setup

We have explained in the previous section that each of the rescheduling schemes may cause a delay during the execution of the program in the presence of a permanent fault in the data path. The most important question we want to answer with our experimental setup was: How big is this delay for a typical application that runs on a static scheduled processor that has a permanent fault in one of its operators?

To answer this question, we have investigated several schedules for several benchmark programs, each of them running on an application specific VLIW architecture. The benchmark programs are inner loop kernels, which are executed frequently. The VLIW architectures were adapted by a design space exploration [27] to the considered application. This means that there are no spare operators in the data path. Every available operator is really needed for the execution of the program, and a fault in one of them must cause a delay. Table 1 shows some characteristics of the used benchmark schedules.

Name	Length	Add	Sub	Mul	Nop
DCT-DIT	9	24	12	12	6
DCT-LEE	12	17	11	21	11
DCT-DIF	8	17	12	12	7
DCT-DIF	9	17	12	12	4
DCT-DIF	11	17	12	12	3

Table 1: Characteristics of the used benchmark schedules.

The schedules are the inner loop of different discrete cosine transformations (DCT). The number of instructions into which the operations were scheduled, i.e. the length of the schedule, is shown in the column *Length*. The number of operations of a certain type in each schedule is shown in the columns *Add*, *Sub*, *Mul* and *Nop*. The characteristics of the VLIW architectures that were adapted to each of these schedules are shown in table 2.

Name/L	FU1	FU2	FU3	FU4	FU5	FU6
DCT-DIT/9	–	+ *	– *	+ *	+	+ *
DCT-LEE/12	+ *	–*	+	+ *	– +	n.a.
DCT-DIF/8	– +	+ *	–	+	– *	–
DCT-DIF/9	– +	– + *	–	+	+ *	n.a.
DCT-DIF/11	– +	– + *	– *	– +	n.a.	n.a.

Table 2: Characteristics of the VLIW-architectures that were adapted to the corresponding schedules in table 1.

Every column *FUx* shows the operators that are included into the FU *x*. Thereby, `n.a.` means that the architecture does not include the corresponding FU. More characteristics of the used benchmarks and corresponding architectures can be found in [26].

We estimated the delay related with each rescheduling scheme under the assumption that one operator is faulty. For each possible fault of such a type in the architecture and each rescheduling scheme, we estimated the delay during the execution of the program by the following rules that are derived from the behaviour explained in section 3.1: We assume a fault in operator *x* of FU *a*.

- In the simple hardware solution, a delay of one occurs for each instruction in the schedule, where an operation of type *x* is executed on FU *a*.
- In the complex hardware solution, a delay of one occurs only for those instructions in the schedule, where an operation of type *x* is executed on FU *a* and all other FUs that include an operator of type *x* execute an operation that is not a NOP.
- In the software solution, a delay of one occurs for those instructions in the schedule, for which no

permutation of the operations exists such that FU *a* executes an operation different from type *x*.

The worst case fault is the fault of an operator for whose fault the sum of the delays for all instructions in the schedule is maximal according to the rules above. This delay is called the worst case delay. In the same way the best case delay is defined. It is the minimal delay that occurs for a possible fault in the data path.

From these considerations we excluded the time for detecting a permanent fault and also the time for rescheduling the application by the software-based approach. This is done, because we assume that for all three rescheduling schemes the fault detection must be done off-line, i.e. the system is not in use. Thus, there will be also some additional time for executing the software-based rescheduling algorithm. Furthermore, we do not add this rescheduling time to the execution time of the software rescheduled program, because it is a fixed amount of time. The estimated delay for the execution of the inner loop kernel appears in each loop iteration. Thus, by executing enough iterations, the constant amount of time for software based rescheduling becomes negligible small.

5 Results

Table 3 shows the worst and best case delay in percentage of the original execution time of the corresponding schedule (the number before /). Each operator may be faulty and represents a possible fault case. In each column the number behind the / is the number of fault cases where a faulty operator leads to the worst/best case delay.

The results show that in one case the execution time of the corresponding application is increased by the simple hardware solution by 91%. For the same schedule, the execution time is increased by the complex hardware solution by 82%. The software solution is able to find much more freedom in the given schedule and increases the execution time only by 36%. In general, the results show that the simple hardware solution produces the highest execution overhead (between 90% and 100%), which is no surprise. In most cases the complex hardware solution is better (approximately 10% faster) than the simple hardware solution, because most schedules contain some instructions with NOPs. The better performance is at the price of a more complex control path. The software solution outperforms both approaches (approximately 10% faster then the complex hardware solution). It finds freedom for rebinding even in instructions without NOPs. Moreover, it avoids a complex control path that is error-prone itself. The control path is almost the same as the one for the simple hardware solution. But by software-based rescheduling the worst-case delay can be decreased by approximately 20%.

6 Conclusions

Our results show that the worst case delay of a schedule can be reduced by approximately 20% with our proposed software-based rescheduling, compared to the simple hardware rescheduling. However, the results also show that there is still a huge delay compared with the execution time in the non-faulty data path, which is a critical point especially for hard real-time applications. If it is not permitted to increase the original execution time in the case of a permanent fault, either more operators must be included in the data path to give more freedom to the rescheduling algorithm, or another schedule with less resource utilization must be used that is adapted to the occurred fault by software-based rescheduling. That schedule implements the same function as the original schedule, but it computes the results with less accuracy [24]. If the application tolerates the reduced accuracy for a certain time, which can be the case for example in image and audio processing applications, this is an alternative to redundant hardware.

Unfortunately, the software solution still needs the simple hardware solution as backup rescheduling mechanism for the case that there is no valid permutation of the operations in an instruction. This disadvantage may turn out to be an advantage because it provides a simple way to improve the performance only in parts of the schedule, i.e. in inner loops. Less frequently executed parts of the application are not optimized by the software rescheduling algorithm. There the hardware support is used automatically. This means, the software-based rescheduling can be also understood as an optimization of the executed program in the field. Delays which are introduced by the simple hardware solution are minimized by the optimization (i.e. the software-based rescheduling). Moreover, this optimization must not be done immediately after the detection of a permanent fault. The executed program can rather be optimized when, the system is not in use for a short time or concurrently, if a test processor is available.

Rescheduling	**DCT-DIT/9**		**DCT-LEE/12**		**DCT/DIF8**		**DCT/DIF9**		**DCT-DIF/11**	
Scheme	**Worst**	**Best**	**Worst**	**Best**	**Worst**	**Best**	**Worst**	**Best**	**Worst**	**Best**
SHS	89% / 2	11% / 1	83% / 1	17% / 1	100% / 1	13% / 1	100% / 1	11% / 2	91% / 1	9% / 1
CHS	67% / 3	11% / 1	58% / 2	8% / 1	100% / 1	0% / 1	89% / 1	11% / 2	82% / 1	0% / 1
SWS	67% / 1	11% / 3	50% / 2	0% / 3	88% / 1	0% / 1	78% / 1	0% / 2	36% / 2	0% / 3

Table 3: Worst and best case delay for the benchmark schedules.

References

[1] *ITRS Roadmap - Design*: 2005.

[2] R. I. Bahar, M. B. Tahoori, S. K. Shukla and F. Lombardi: *Challenges for Reliable Design an the Nanoscale*. IEEE Design & Test of Computers, 22(4), pp. 295-297, 2005.

[3] R. Baumann: *Soft Errors in Advanced Computer Systems*. IEEE Design & Test of Computers, 22(3), pp. 258-266, 2005.

[4] A. Benso, S. D. Carlo, G. Di Natale and P. Prinetto: *Online Self-Repair of FIR Filters*. IEEE Design & Test of Computers, 20(3), pp. 50-57, 2003.

[5] M. A. Breuer, S. K. Gupta and T. M. Mak: *Defect and Error Tolerance in the Presence of Massive Numbers of Defects*. IEEE Design & Test of Computers, 21(3), pp. 216-227, 2004.

[6] Y.-Y. Chen, H. Shi-Jinn and L. Hung-Chuan: *An Integrated Fault-Tolerant Design Framework for VLIW Processors*. 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), pp. 555-562, 2003.

[7] A. DeHon and H. Naeimi: *Seven Strategies for Tolerating Highly Defective Fabrication*. IEEE Design & Test of Computers, 22(4), pp. 306-315, 2005.

[8] P. E. Dodd and L. W. Massengill: *Basic Mechanisms and Modeling of Single-Event Upsets in Digital Microelectronics*. IEEE Transactions on Nuclear Science, 50(3), pp. 583-602, 2003.

[9] C. Galke, M. Pflanz and H. T. Vierhaus: *A Test Processor Concept for Systems-on-a-Chip*. Int. Conference of Computer Design (ICCD'02), 2002.

[10] C. Galke, H. Schwabe, H. Fröschke, et. al.: *Processor Design for Functional Self Test: A Strategy and it's Limits*. Dresdner Arbeitstagung für Schaltungs- und Systementwurf (DASS'05), 2005.

[11] L. Guerra, M. Potkonjak and J. M. Rabaey: *High Level Synthesis Techniques for Efficient Built-In-Self-Repair*. IEEE Workshop on DFT in VLSI systems, pp. 41-48, 1993.

[12] S. Habermann, R. Kothe and H. T. Vierhaus: *Built-in Self Repair by Reconfiguration of FPGAs*. 12th IEEE International On-Line Testing Symposium (IOLTS 2006), pp. 187-188, 2006.

[13] R. Karri, K. Hogstedt and A. Orailoglu: *Computer-Aided Design of Fault-Tolerant VLSI Systems*. IEEE Design & Test of Computers, 13(3), pp. 88-96. 1996.

[14] R. Karri, K. Kim and M. Potkonjak: *Computer Aided Design of Fault-Tolerant Application Specific Programmable Processors*. IEEE Transactions on Computers, 49(11), pp. 1272-1284, 2000.

[15] M. E. Kavanaugh: *The Twenty-Fifth International Symposium on Fault-Tolerant Computing - Highlights from 25 Years*. IEEE Computer Society Press, 1995.

[16] I. Koren and C. M. Krishna: *Fault-Tolerant Systems*. Morgan Kaufmann, 2007.

[17] R. Kothe, C. Galke, S. Schultke, et. al.: *Hardware/Software Based Hierarchical Self Test for SocS*. 9th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS'06), pp. 159-160, 2006.

[18] R. Kothe and H. T. Vierhaus: *Repair Functions and Redundancy Management for Bus Structures*. Workshop on Dependability and Fault Tolerance at ARCS'07, 2007.

[19] R. Kothe, H. T. Vierhaus, T. Coym, et. al.: *Embedded Self Repair by Transistor and Gate Level Reconfiguration*. Proc. of the IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'06), pp. 210-215, 2006.

[20] P. K. Lala: *Self-Checking and Fault Tolerant Digital Design*. Morgan Kaufmann, 2000.

[21] M. Mishra and S. C. Goldstein: *Defect Tolerance at the End of the Roadmap*. Proc. of the IEEE Int. Test Conference, pp. 1201-1210, 2003.

[22] S. Mitra, W.-J. Huang, N. R. Saxena et. al.: *Reconfigurable Architecture for Autonomous Self Repair*. IEEE Design & Test of Computers, 21(3), pp. 228-240, 2004.

[23] S. Mitra, W.-J. Huang, N. R. Saxena et. al.: *Reconfigurable Architecture for Autonomous Self-Repair*. IEEE Design & Test of Computers, 23(3), pp. 228-240. 2004.

[24] P. Pawlowski and M. Schölzel: *A Case-Study for Built-In-Self-Repair in Application Specific Processors By Decreasing the Arithmetic Accuracy*. Proc. of the IEEE Workshop Signal Processing'2006, pp. 77-82, 2006.

[25] D. Scheit: *Built-in self-repair of interconnect strucures on system-on-a-chip*. Computer Science Report 03/07, pp. 31-36, 2007.

[26] M. Schölzel: *Automatisierter Entwurf anwendungsspezifischer VLIW-Prozessoren*. Dissertation, BTU Cottbus, 2006.

[27] M. Schölzel and P. Bachmann: *DESCOMP: A New Design Space Exploration Approach*. Proc. of the 18th Int. Conference on Architecture of Computing Systems (ARCS'05), pp. 178-192, 2005.

[28] S. Shoukourian, V. Vardanian and Y. Zorian: *SoC Yield Optimization via an Embedded-Memory Test and Repair Infrastructure*. IEEE Design & Test of Computers, 21(3), pp. 200-207, 2004.

[29] S. Sirisantana, B. C. Paul and K. Roy: *Enhancing Yield at the End of the Technology Roadmap*. IEEE Design & Test of Computers, 21(6), pp. 563-571, 2004.

[30] Y. Zorian and D. Gizopoulos: *Design for Yield and Reliability (Guest Editor's Introduction)*. IEEE Design & Test of Computers, 21(3), pp. 177-182, 2004.