A Reconfigurable Generic Dual-Core Architecture

Thomas Kottke[1]
EADS Deutschland GmbH
D-88090 Immenstaad
Email: thomas.kottke@eads.com

Andreas Steininger
Vienna University of Technology
A-1040 Wien, Treitlstrasse 1-3
Email: steininger@ecs.tuwien.ac.at

Abstract— **In this paper we propose a generic frame for the implementation of a dual-core processor with two modes of operation. One is the safety mode that allows to run the two cores in lock step in a classical master/checker fashion. A clock delay of 1.5 clock cycles between master and checker establishes the temporal redundancy to minimize the potential for common mode faults. The second operation mode allows a parallel execution of different instruction streams on the two cores in a multiprocessor fashion. The possibility to dynamically switch between the two modes allows for an efficient utilization of the duplicated core.**

We propose an implementation of such a generic frame that can be applied in conjunction with virtually any standard processor core. Also we perform a systematic failure analysis for the safety mode and the mode switching procedure. Experimental fault injection confirms that our reconfigurable architecture indeed provides the same fail safe properties as the classical master/checker architecture.

I. Introduction

In future automobiles more and more complex microprocessor-based control systems will be implemented for safety-critical applications such as antilocking brake systems, electronic stability program or x-by-wire systems. To meet the high safety requirements of future applications – as demanded in the European standard EN 61508, for instance – in spite of the increasing rate of transient errors [1], [2] that is predicted to result from reduced voltage levels and shrinking feature size [3], the microprocessors have to be equipped with powerful mechanisms for error detection and error handling. While it is relatively easy to protect memory or communication interfaces by means of coding techniques, e.g., the core is more difficult to protect (see [4], [5], [6], [7], [8], [9] for example). One attractive generic solution in this context is a dual-core (master/checker, e.g. in [10]), because it is quite easy to implement with standard cores and – as shown in [11] – exhibits a very high error detection coverage.

At the same time there is an increasing demand for computing power. While this demand is usually satisfied by ever increasing clock rates in office applications, clock frequency is constrained in automotive applications, and hence architectural solutions are sought to improve performance while keeping clock rates moderate. In this context multiprocessor architectures are an attractive solution.

In a highly competitive market like the automotive the most demanding requirement for microprocessors is low cost, and unfortunately both, master/checker architectures and multiprocessor architectures tend to make the system expensive. There is, however, a potential for an elegant combination of these two approaches: Control systems usually not only perform safety critical tasks such as the algorithm to control safety-relevant sensors and actuators, but also uncritical calculations dedicated to comfort functions, for instance. This led to the essential idea of switching resources between safety and performance, which was derived in an internal research of Bosch [12] and led us to consider a reconfigurable dual-core system that is capable of handling both demands – safety and high computing power – while making efficient use of the duplicated core. For the safety-critical tasks the system is configured as a master/checker, while the non-critical tasks are computed in a multiprocessor fashion. For this approach to be reasonable, (a) the switching between the modes must be possible with low overhead, and (b) the additional logic for mode switching must not compromise fault tolerance in the master/checker operation. Especially issue (b) will be a main focus of our paper.

To satisfy the demand for low cost and high efficiency it is further necessary that the performance of the processor is tightly matched with the application requirements. Therefore a wide range of processors with different performance must be considered. It is, however, very expensive to adapt an existing concept and repeat the safety verification for every single core used. With this motivation we propose a *generic* framework here that treats the processor cores as black boxes with some general assumptions on their input/output behavior. Our aim is to verify the properties of this generic architecture such that our results are valid independent of the actual type of processor core that is embedded into our frame. We do not consider diversity of cores or software, since we target hardware faults that occur during operation, rather than design and software faults that can be eliminated through extensive testing.

There is a lot of literature investigating the use of the master/checker approach for safety relevant applications [13], [14] and the potential of the multiprocessor approach for improving performance [15], [16]. None of these, however, considered reconfiguration to facilitate switching between the two modes. Switching between different fault-tolerant modes in software at the operating system level was proposed in [17].

This paper starts with an introduction to the proposed reconfigurable dual-core system in chapter II. Its implementation will be outlined in chapter III, while chapter IV is concerned

[1]This work was performed while the author was with Robert Bosch GmbH, Germany.

with a theoretical fault analysis of the system. Subsequently the setup of the fault injection experiments will be sketched in chapter V and the experimental results be presented and discussed chapter VI. Finally, chapter VII concludes the paper.

II. Operation of the split core frame

In order to meet the high efficiency demands our proposed reconfigurable dual-core system allows to switch between two modes: For safety critical applications the *safety mode* can be chosen, in which the system is configured in the classical master/checker fashion. To make our approach acceptable we have to prove that it indeed provides the same fail safe capabilities as a conventional master/checker system. In the *performance mode* we operate our architecture like a dual processor system. Our aim here is to attain the computing power of a two-processor system.

The switching between the two modes occurs under software control. We have reserved a special instruction – in the following referred to as the *mode switching instruction* – for triggering the alternation between the two modes. This mode switching instruction is recognized by a core-external mode switch unit, while it is transparently handled as a no operation instruction within the core.

A. Safety Mode

In this mode core 1, the master, is directly controlling the peripherals such as memory or actuators. Core 2, the checker, receives the same instruction stream as core 1 and is hence performing the same operations (usually in lock-step). The checker's outputs, however, are not connected to peripheral components, but are used for checking the correctness of the respective master outputs. This output-based comparison scheme has proven to exhibit a very good detection coverage for core-internal errors. The inputs (control signals and instruction stream) that are used by both cores in common, however, are not covered by this comparison, therefore their protection deserves special attention. The same is true for the outputs after comparison. To harden the system against common mode failures induced for example over the power supply or by electromagnetic interference the two cores are operated with a temporal displacement of 1.5 clock cycles as suggested in [24]. Consequently the master's outputs have to be delayed by 1.5 clock cycles before they can be compared with the checker's outputs.

B. Performance Mode

In performance mode the two cores execute different instruction streams in parallel and each core can control the peripherals. In order to provide efficient access to the (synchronous) memories and peripherals, both cores have to operate with the same clock polarity now. Therefore the temporal displacement of the checker clock is disabled in performance mode. Simultaneous requests from both cores to the same memory area have to be explicitly resolved by special units (instruction RAM control unit, data RAM control unit). To prevent the instruction memory from becoming a performance bottleneck each core is provided with an individual instruction cache. A block size of 4 instruction words allows an efficient burst access yielding low mutual deceleration of the cores. Since in automotive applications only about every tenth instruction is an access to data memory, we did not implement a data cache. Should the proportion of data memory accesses increase in future applications a data cache can easily be supplemented.

C. Switching between the modes

In performance mode the cores are operating independently from each other on different instruction streams. In safety mode they execute the same instruction stream and are assumed to behave identically. This requires that their internal state is identical, in particular register and cache contents must be the same on both cores. So obviously some kind of data synchronization must be performed before switching from performance mode to safety mode. The responsibility for synchronizing the register contents can quite easily be moved to the operating system. Establishing consistency between the caches, however, requires dedicated hardware support. Should one core encounter a cache miss while the other has a hit for the same instruction access, the temporal behavior of the two cores would diverge causing a comparator mismatch and hence a false error indication. The simplest remedy here would be a complete cache flush at every transition from performance mode to safety mode. This would, however, unnecessarily degrade cache utilization and hence waste performance. As an improved strategy we have implemented a list of flags indicating the validity of every cache line in safety mode. The flag is set valid if the associated cache line is loaded in safety mode. Should the cache line be reloaded in performance mode by any of the two cores, the flag is reset to invalid. Using this information in safety mode we can easily judge whether a given cache line is consistent in both cores. Further improvements to the caching strategy are possible, of course, but not within the scope of this paper.

With these provisions the performance penalty for switching between the modes is very low (in the following we will show that switching indeed occurs very fast). As a result, in safety mode the performance of our reconfigurable dual-core system meets that of a single core. In performance mode the actual performance depends on cache hit rate and frequency of data memory accesses. We are currently trying to assess the respective performance figures in a systematic manner. This paper, however, will focus on the fail-safe properties of the proposed reconfigurable architecture.

III. Implementation of the split core frame

The proposed framework is based on the assumption of a Harvard architecture for the processor cores used. Knowing that most processors practically used in automotive applications exhibit a Harvard architecture this is a reasonable restriction. The concrete processor we use to test our framework and to demonstrate its functionality is called SPEAR [18], [19]. This processor was selected, because it provides all

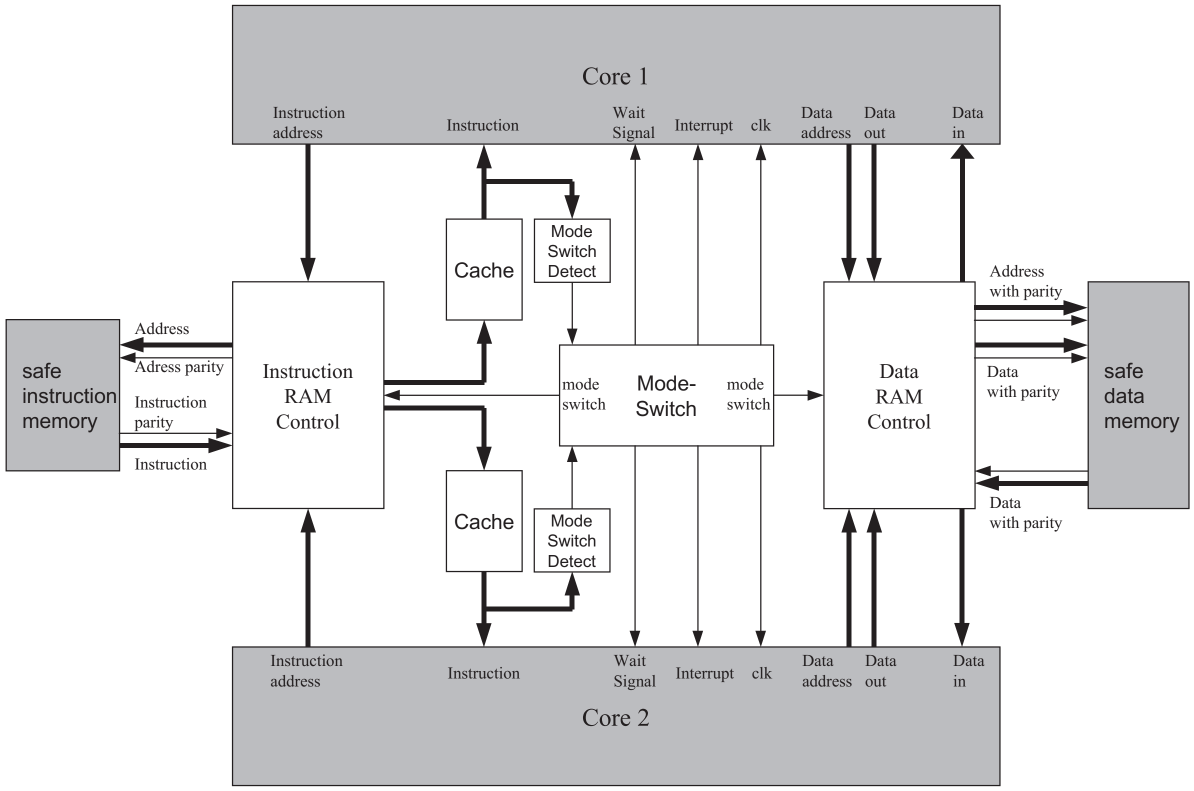

Fig. 1. Block diagram of the proposed reconfigurable dual core architecture

features found with state of the art processors. It is a RISC processor with Harvard architecture, has memory mapped IO and is able to meet the temporal demands of real-time systems. Although today automotive processors normally have 32 bit bus width, we considered the 16 bit architecture of SPEAR satisfactory for our study of the specific implementation problems of the reconfigurable dual-core systems. While this choice does not change the fault tolerance properties we intended to investigate, the resulting reduction in hardware complexity allowed us to implement the system in an FPGA and furthermore yielded shorter simulation time for the fault injection experiments. Figure 1 shows the implementation of the reconfigurable dual-core system.

The components of the frame are shown as transparent boxes while the embedded standard components (processor cores and memories) are shaded grey. Notice that we do not duplicate the expensive memory components but employ non-redundant ”safe memories” as proposed in [20] instead. The main components of the reconfigurable dual-core frame are the mode switch detect units, the mode-switch unit and the control units for instruction RAM and data RAM. In the following we will briefly sketch the functionality of these units.

A. Instruction RAM control unit (ICU):

The ICU handles all accesses of the two cores to their common instruction memory. In safety mode core 1 exclusively supplies the instruction address (in case of a cache miss). In response the ICU not only fetches the requested single instruction, but automatically performs a burst access that loads the complete 4 word cache block from the instruction RAM. This block is directly passed through to the cache of core 1, while core 2’s cache is provided the same block with a delay of 1.5 clock cycles.

Since in performance mode both cores request instructions independently, the ICU has to resolve simultaneous requests. In general core 1 has priority over core 2. To avoid starvation, however, core 2 is given priority when core 1 has already accessed the instruction memory in the previous cycle.

B. Data RAM control unit (DCU):

The DCU handles accesses of the two cores to the peripherals and the data memory they share. In addition it has to provide an individual identification bit for each core. This information is crucial for identifying a core’s individual role within the schedule in performance mode. This core identification bit can be read by each core at a reserved memory address. While this address is the same for both cores, core 1 will read a 0 and core 2 a 1.

In safety mode all accesses to data memory and peripherals are exclusively performed by core 1, while the requests from core 2 are used for comparison only. Read data are fed to core 1 directly and to core 2 with the 1.5 clock cycle delay.

In performance mode the DCU has to prioritize simultaneous requests to data memory or peripherals from both cores.

Basically the same arbitration scheme is implemented as in the ICU. In addition a semaphore mechanism is available for locking the data memory (similar to the MESI-protocol): One core can lock the data memory for exclusive use, while any attempt by the other core to access the memory will be rejected by the DCU until the memory is unlocked by the first core. Locking and unlocking is performed by accessing the specific memory location that is recognized by the DCU. The same priority scheme applies as above.

We have moved this memory management function into the DCU to relieve our standard cores from this requirement.

C. Mode switch detect unit:

The mode switch detect units are each snooping the instruction bus between the cache and the core. As soon as they detect the mode switch instruction, they trigger the mode switch unit. This functionality could easily be moved to the cores' instruction decoders. In our implementation, however, we preferred the employment of external units, since one of our initial aims was to use standard cores without any modifications. A disadvantage of our implementation is that the switch instruction becomes effective as soon as it is fetched. Should a preceding jump change the program flow, the switch instruction is still effective, although it is in the branch delay slot and hence will be flushed from the core's pipeline. As a consequence the instructions at the branch target will be executed in the wrong mode. This problem, however, is easy to solve by standard compiler techniques like instruction reordering.

D. Mode switch unit:

As already mentioned the mode switch is performed under software control, while the mode switch unit provides the necessary hardware support. The following code sequence illustrates the switch procedure from safety mode to performance mode:

```
LDL       r1, 248
LDH       r1, 255       (1)
MODE SWITCHING          (2)
LDW       r2,r1         (3)
BTEST     r2, 1         (4)
JMPI_CT                 (5)
```

In line (1) register r1 is loaded with the address of the register location within the DCU that holds the core identifier bit. Next (2) the mode switching instruction is executed. Since the two cores are working in safety mode with a delay of 1.5 clock cycles, core 1's mode switch detect unit is the first to recognize the mode switching operation. In response it issues a trigger (`core1_signal`) to the mode switch unit which in turn halts core 1 by activating the signal `wait1`. 1.5 clock cycles later core 2's mode switch detect unit recognizes the mode switch instruction. The mode switch unit now halts core 2 for half a cycle and aligns the clocks of the two cores in phase. Next it changes the mode bit from safety mode to performance mode and lets the two cores precede with their operation – with now identical clocks. In the next step (3) both cores load their core identifier bit from their respective DCU. In line (4) this bit is checked and a conditional branch is executed by core 2 in (5), because its identifier bit is 1. Core 1 will not take the branch, since its core identifier bit is 0. This finally causes the control flow of the two cores to diverge as intended. The timing diagram for this procedure is shown in Figure 2.

The right part of Figure 2 also shows the switch from performance mode to safety mode, during which the individual instruction flows of the two cores have to be merged. In the example shown core 1 is the first to encounter a mode switch instruction within its instruction stream. The mode switch detect unit indicates this to the mode switch unit by activating `core1_signal`. Like above the mode switch unit reacts by halting core 1. In addition it activates the signal `message2` now, thus triggering an interrupt at core 2. In response core 2 executes a routine to save its context and then jumps to the mode switch instruction at which core 1 is still waiting (the respective jump target is well known by the scheduler of the operating system). Core 2 then also executes the switching instruction upon which its mode switch detect unit also triggers the mode-switch unit. At this point the mode switch unit halts core 2 while letting core 1 start working again. It inverts the clock for core 2 and lets core 2 start as well, but with the desired delay of 1.5 clock cycles. After having initialized the registers appropriately the two cores are finally synchronized for the safety mode.

IV. Theoretical failure analysis

In accordance with the chosen duplication and comparison approach in safety mode the primary aim we want to achieve is fail-safe behavior under the single-fault assumption. For this purpose we must be able to prevent that erroneous data are propagated to external components at any time. In particular we want to prove that in safety mode our proposed reconfigurable frame performs as well as the traditional master/checker architecture in this respect, or, in other words, that the addition of the switching capability does not compromise the fail-safe property. In this section we will therefore systematically analyze our architecture for single points of failure. In this analysis we will distinguish four different areas to protect: the interfaces, the memories, the cores and the non-duplicated components within the frame.

A. Memories

The duplication of memories would be advantageous for performance mode as well as for safety mode. Considering, however, that the memory usually is by far the largest – and hence most expensive – part of the processor and is easy to protect by means of coding, a duplication is a very costly and inefficient solution. Therefore we have decided to use a single data memory and a single instruction memory that are both shared by the two cores. We have developed a special "safe memory" that is equipped with sufficient error detection and self-testing capabilities to tolerate any single fault. These

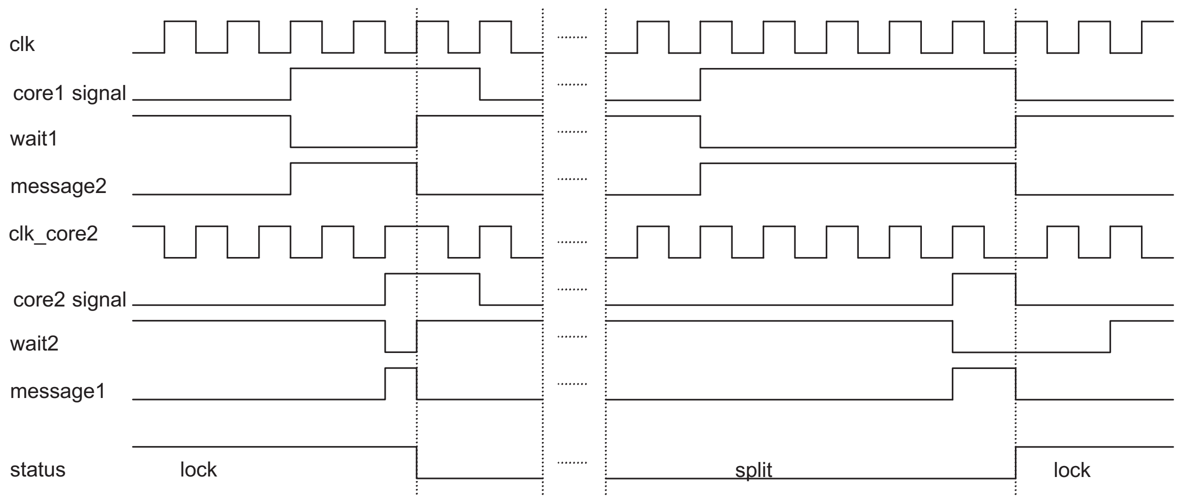

Fig. 2. Timing diagram illustrating the mode switching

memories will not further be detailed here; the interested reader is referred to [20].

A substantial problem for the data memory is retaining the integrity of safety relevant data while the core is operating in performance mode. Since the cores are not protected in performance mode, their operation is no more fail safe and they might well perform erroneous accesses to external components. While this behavior can in general be accepted during non-critical calculations, provisions must be made to prevent the pollution of safety relevant data. Such a service is usually provided by a memory management unit (MMU) that permits access to a sensitive memory area in supervisor mode only. However, demanding that the cores must be equipped with an MMU would severely limit the choice of applicable cores, therefore we have decided to move the required functionality to our safe memory. The memory simply uses a dedicated `core mode` signal issued by the mode switch unit to distinguish, whether the core is operating in performance mode or in safety mode. Provided with this information the memory can enforce a write protection for a specific area during performance mode and hence effectively protect safety relevant data from being erroneously overwritten. The same solution can be applied for external peripherals holding safety relevant data.

Obviously this solution relies on the integrity of the `core mode` signal. Therefore this signal is protected by dual-rail coding (see below). Furthermore we must ensure that there is no erroneous mode change from performance to safety mode and that no erroneous assignment of `core mode` as safety mode is possible in performance mode (which would allow a task in performance mode to disguise as one in safety mode). These issues will be discussed later.

B. Cores

DCU and ICU are both equipped with internal comparators that check the equality of the inputs from both cores in safety mode (properly considering the 1.5 cycle delay of core 2). Just like in the traditional master/checker architecture this approach effectively detects all faults of a single core as soon as they become effective at the output. Notice that in our approach the caches – although not considered part of the processor core but implemented separately within the frame – are also included in this protection, since they are in the signal path from processor core to ICU.

The *comparators* are not duplicated, therefore they have been designed as totally self-checking components as proposed in [21], [22]. If the comparison of the output signals is time-critical, an approach for the comparator as shown in [23] can be used. All signals on the output buses are compared: data-lines, address-lines and the incoming and outgoing control signals to and from the external modules. Obviously the comparator output is meaningful in safety mode only.

For economic reasons both cores are located on the same die. This causes the potential for common mode failures, especially in response to faults on the (common) clock or the (common) power supply. According to the results presented in [24] time diversity is a very efficient means to defeat these types of common mode failures. Therefore we are operating core 2 with a delay of 1.5 clock cycles in safety mode. Still, however, we have to take a closer look at the specific fault types on clock and power supply lines:

Clock: Hypothesized failures of the clock signal are a totally missing clock and a partially missing clock (i.e. for some components). Failures in the clock generator resulting in pulse omissions are not considered here, because they should be solved in context with the clock generator concept. Transients on the clock line are common mode failures and should hence be covered by the time diversity.

As shown in [25] a missing clock signal or a severely "hung-up" core is difficult to discover by an internal mechanism, hence an external reference like a watchdog is necessary. Consequently one of the two cores can be assigned the responsibility to trigger the watchdog. This, however, does not provide complete coverage of all partial clock failures unless the watchdog is triggered by the `core mode` signal:

A generic structure of the global clock net that distributes the clock among the components of our system is shown in Figure 3. Imagine the case that the clock line is disrupted at position (2). This will cause core 1 to continue working, while all other components will stop. The data frozen in the delay component and the data frozen in core 2 will probably be the same, therefore the comparators will never activate their error signal, even in case the master should produce an erroneous output. If core 1 were assigned the task to toggle the watchdog, both the watchdog and the comparator would fail to detect this error.

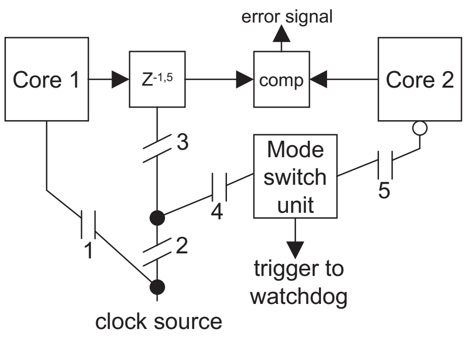

Fig. 3. Possible fault locations in the clock tree

A much better solution is to trigger the watchdog with the `core mode` signal issued by the mode switch unit. As explained above, a mode switch requires a trigger by both cores. If the clock line is broken at position (1) core 1 will stop working, which further inhibits any mode change. For disruptions at positions (2), (4) or (5) core 2 will stop its operation, which again blocks any further mode change. The missing mode change will eventually allow the watchdog to detect the error. The same is, of course, true for a totally missing clock, provided that the watchdog is timed from a source independent of the clock to be checked. Finally, a disruption at position (3) will cause a malfunction of the delay component, which will be detected by the comparator in safety mode. To force the triggering of the watchdog when running an application continuously in safety- or performance mode, the programmer has to insert "dummy" mode switches. Since these do not occur too frequently and are executed very fast, their overhead will be negligible. In summary the proposed strategy ensures that all partial clock failures are covered.

Power supply: Disruption in the power supply will either result in a total stop of the system or – in case of spikes or transient outages – in a temporary disruption of the operation of one or more components. Again the time diversity ensures that the disruption will cause different effects in the two cores. Moreover, such types of failures tend to result in a program flow disruption, such that the mode switch unit will very likely not trigger the watchdog correctly. If only one component is disrupted, the comparators will detect the failure in safety mode. Low supply voltage of the comparators will also be detected, because the dual-rail signals will not be able to display a proper high logic level. In summary we can expect all power supply failures to be detected in safety mode, while failures are likely to occur in performance mode.

The *mode switch detect units* are also duplicated . Unlike the caches, however, their behavior is not checked by a comparator. As will become clear later on, an erroneous behavior of the mode switch detect units (i.e. false or missing requests) can be detected by the mode switch unit.

C. Interfaces

Buses: The buses for the incoming and outgoing data and instructions are quite easy to protect against single faults by means of parity. In order to ensure that in the safety mode at least one core is getting correct data – otherwise the bus would constitute a single point of failure – the buses for the incoming data are routed as shown in Figure 4.

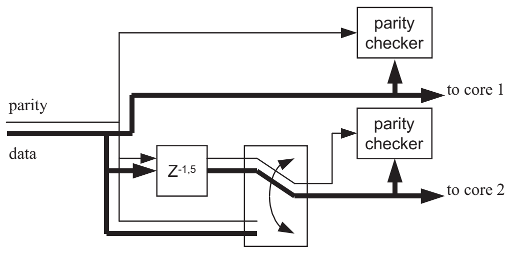

Fig. 4. Routing of the buses for incoming data

The parity checkers are implemented totally self checking and their output signal is coded as an alternating dual-rail signal to make stuck-at-inactive behavior of the parity checker detectable. A multiplexor routes the data to core 2 over a 1.5 clock cycles delay element in safety mode and feeds the data directly to core 2 in performance mode. Under the single-fault assumption this multiplexor need not be explicitly secured, since its failure will lead to a comparator mismatch in safety mode. (In performance mode a faulty multiplexer might erroneously delay the data, which will decrease the performance of core 2.)

The proposed implementation of the data path for outgoing data is shown in Figure 5. Here it is important to generate the parity for the outputs of both cores individually and include it in the comparison. This way the two parity generators are checking each other and hence do not require additional protection. Any single fault on the data path beyond the checker can safely be detected by a parity check at the destination. Some types of single faults in the multiplexor might cause multiple faults on the data bus (incorrect switching, e.g.). Therefore the multiplexor implementation needs special attention (for details see [26]).

Single-bit inputs: For the protection of single-bit input signals like *interrupts* or *reset* dual-rail coding is employed. After having passed through an input synchronization stage,

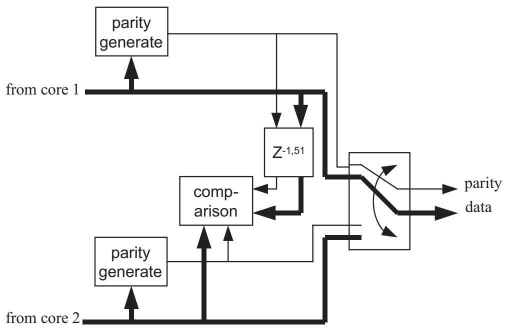

Fig. 5. Routing of the buses for outgoing data

the non-inverted rail of such a signal is routed to core 1, while the inverted rail goes to core 2. Care should be taken to perform the re-inversion of the rail at core 2 only after the delay unit, where the temporal redundancy already provides protection. At this point common mode failures caused by electromagnetic interference, for instance, would either result in the two cores executing the interrupt or the reset at different points in time (relative to their respective program flow) or only one executing an interrupt or a reset at all. Both these types of diverging control flow can be easily detected by the comparators.

In performance mode the delay for the core 2 interrupt is disabled. The reset signal for core 2 is always delayed, since after a reset the reconfigurable dual-core system always restarts in safety mode.

Single-bit outputs: All internal error detection mechanisms have to indicate their *error status* by an alternating dual-rail signal as shown in [27] and [28]. In case a mechanism detects an error, its output goes to “00” or “11”, while the output is “01” or “10” otherwise. This coding is also applied for the `core mode` signal. It allows an evaluation by a dual-rail comparator. In order to detect stuck-at faults on a single rail early, the signal has to alternate with every check performed, i.e. even if the status remains unchanged the other semantically identical codeword is used. It is important to retain the dual-rail encoding beyond the chip outputs, since otherwise the output pins are unprotected.

D. Non-duplicated components

Mode switch unit: The mode switch unit must ensure that the safety critical procedure of switching between the two modes is executed correctly. For this purpose (1) its implementation must be appropriate for the single-fault assumption, and (2) the mode switch procedure implemented within this unit must be able to cope with (single) erroneous inputs. With respect to (2) we have to further distinguish two cases: (a) An erroneous request for a mode switch is issued by one side, or (b) a scheduled mode switch request from one side is erroneously suppressed by the other side. In both cases the reason may be a single fault in one of the cores, the associated mode switch detect unit or the software.

As already outlined in the previous section, a request from both cores is required for a mode switch to be actually executed. Therefore case (2a) ends up with the erroneous core being blocked by its own request, while the fault-free core continues working properly. In safety mode this situation will lead to a comparator mismatch, while in performance mode the erroneous core’s task(s) will simply not further be executed, which degrades a non-critical service. In case (2b) the fault-free core is unnecessarily blocked by the faulty core, but only until the watchdog times out (remember that an actual mode change is required to trigger the watchdog). Since errors in the program flow of one core tend to cause inappropriate/missing mode changes, the mode change mechanism has the potential for detecting control flow errors.

Let us consider the implementation of the mode switch unit now (case (1) from above). One main duty of the mode change unit is to provide the appropriate clock for core 2 depending on the current mode. In safety mode any failure in the clock is detected by the comparators, because the well-defined temporal relation between master and checker becomes upset. The other main duty of the mode switch unit is to synchronize the operation of the two cores upon a mode switch. Failure of this service will again upset the intended temporal relation in safety mode and hence be detected by the comparators. Finally the mode switch unit has to generate the `core mode` signal which has a critical functionality for memory access and triggering the watchdog. As already mentioned this signal is dual-rail coded (and also generated in a dual-rail fashion) to protect it from single faults.

Instruction RAM control unit (ICU): The ICU implements the interface protection for the outgoing instruction address bus and the incoming instruction bus as shown in Figure 4 and Figure 5, respectively. Since parities are checked in every mode, while a comparison of the instruction addresses issued by the cores is performed in safety mode only, an error signal is provided for each mode individually. The appropriate signal (dual rail) is selected by the mode switch unit. In addition the ICU is responsible for generating the address sequence for the burst access. The state machine required for this purpose is implemented for each core separately, such that in safety mode the outputs of these state machines can be compared. The priority handling feature for the memory accesses is used in performance mode only, therefore this functionality is uncritical.

Data RAM control unit (DCU): The DCU implements the interface protection for the incoming data bus and the outgoing data and address buses. Generation and handling of the error signals are the same as in the ICU. In addition the core identification bit is maintained by the DCU. Basically this bit is used in performance mode only and therefore not safety relevant. Since it is memory mapped, however, it is secured by a parity like all other memory words to keep the access regular. Like with the ICU the state machine for burst access is duplicated, while the priority resolution for memory accesses and the semaphore mechanism are not secured, since they are used in performance mode only.

V. Fault injection experiments

In order to validate our theoretical fault analysis we have carried out a series of fault injection experiments on the proposed reconfigurable generic dual-core architecture. Our aim was to figure out whether the architecture was indeed able to detect all single faults in safety mode. In addition, since we obviously cannot expect our system to reliably detect faults while in performance mode, we wanted to study fault propagation from performance mode to safety mode. Finally the effect of faults that affect the switching between the modes should be assessed.

For these experiments the dual-core consisting of two SPEAR cores and the fault tolerant reconfigurable frame was implemented in VHDL. As shown in Figure 6 two instances of this reconfigurable dual-core architecture have been used in the test environment, one as device under test and another as golden device that operates in lock-step with the device under test and serves as a fault-free reference.

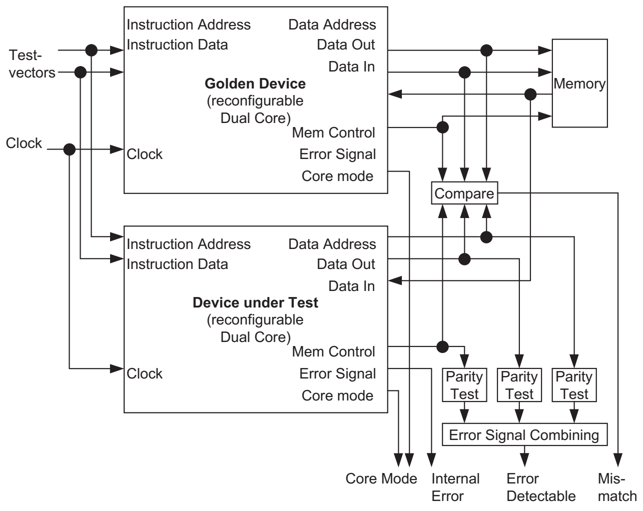

Fig. 6. Fault injection setup

The test environment has been synthesized to a net list in EDIF format (electronic design interchange format). Fault injectors were inserted into this net list at each logic gate input of the device under test. The fault injector is capable of injecting selected stuck-at-one and stuck-at-zero faults at runtime. From this exhaustive list of faults each was activated exclusively once per experiment. We consider the stuck-at fault model sufficient for our purpose of coverage assessment, since our concurrent error detection mechanisms are triggered with an extremely short latency, and it makes no difference whether the fault persists after detection – only the fault effectiveness will turn out higher for permanent faults. Therefore the observed coverage results can be reasonably projected to the bit flip or transient fault model that are anticipated to dominate during field operation.

To attain higher flexibility with respect to the workload we decided to emulate the instruction RAM by the test controller (a PC). Although golden device and device under test are supplied with identical instruction streams, they can access the instructions independently. This allows us to observe the behavior of the device under test even if it deviates from that of the golden device, and hence figure out whether an error is detected and/or has an effect on the workload result later on; e.g. during or after switching the mode.

In particular we needed to observe the following things during the experiments :

1) Has the injected fault become effective? To address this question we compare all signals issued by our device under test to (virtual) external components like the data memory or memory mapped peripherals with the respective reference signals from the golden device. In case of a comparator mismatch (indicated by the signal `mismatch`) we can conclude that the device under test has failed as a result of the injected fault.
2) Has the resulting error been properly detected in time, that is within 2 clock cycles after the error became effective? To address this question we can check wether an error has been detected by a mechanism inside the reconfigurable dual-core framework (signal `internal error`) or is detectable by an external device that employs a parity test on the buses (signal `error detectable`).
3) In which operating mode is the system and is it still working (and triggering the watchdog)? We can check this by observing the `core mode` signal and the sequence of instruction addresses.

By monitoring all the above signals on a cycle-by-cycle base we can assess the temporal relations, in particular we can find out whether error detection occurred before an error had an effect. In order to facilitate a more detailed analysis, we have kept track of whether a fault has been inserted into core 1, core 2 or the frame, and added this information to the respective entry in the observation record.

VI. Experimental results

According to our aims stated above we have executed two experiment series: In fault injection experiment (1) the reconfigurable dual-core system was operated in safety mode only. The selected fault was active all the time. Experiment (2) was aimed at studying the mode switching and the fault propagation from performance mode to safety mode. It started out with a short instruction sequence in safety mode, then a switch to performance mode was initiated with two different applications being executed on the two cores, and finally the workload switched back to safety mode, where the same application was executed as in experiment (1). The fault was activated after having spent a few clock cycles in performance mode.

A. Experiment (1)

Altogether 139632 faults were injected into the reconfigurable dual-core system. 121200 of these faults could be activated by the workload, in the sense that they either had an effect (mismatch with the golden device) or were detected/detectable. This is about 87 percent of all injected faults. In the frame containing the safety critical single components such as the parity checker and the buses 11502 faults were injected. 4275 of these faults could not be activated, because (a) a part of the logic is only used in performance mode and (b) we could not simulate external components such as memory with an address space of 16 bit. Although this means that our experimental results do not cover the full address space directly, we can reasonably assume that these faults would also be detected, because the error detection mechanisms for the involved logic gates are the same as those for the address area that we did cover.

Table I shows the results of experiment (1).

TABLE I
RESULTS OF THE FAULT INJECTION EXPERIMENTS PERFORMED EXCLUSIVELY IN SAFETY MODE

component	master	slave	frame	overall
detected without effect	1029	56962	5334	63325
detected before effect	5026	0	1324	6350
detected during or after effect	50956	0	569	51525
not detected without effect	7055	7102	4275	18432
not detected with effect	0	0	0	0
	64066	64064	11502	139632

Faults in core 1 mostly became effective. The proportion of errors in core 1 that were detected before their effect propagated to the output, depended on the workload. Hence these numbers would look different for a different workload. Errors in core 2 never had an effect on the output. This is no surprise because in safety mode the output of core 2 is only connected to the comparators and not to the outgoing buses. Another interesting observation was that the core mode was correct all the time throughout the experiments. The most important result, however, was the confirmation that all activated failures were detected within a latency of at most 1.5 clock cycles. So with an output delay of 2 clock cycles (as proposed in [26], e.g.) complete error confinement can be ensured. Hence the reconfigurable dual-core system is, when running in safety mode, as safe as a conventional master/checker system.

B. Experiment (2)

The results of the second experiment are shown in Table II. Here also a total of 139632 faults were injected of which 121072 (like before about 87%) could be activated by the workload. 63703 of these faults had an effect in performance mode, but only 28662 were detected during operation in performance mode. When the system switched back to safety mode all of the residual faults were detected but for 458. A closer analysis revealed that these 458 faults affected logic that was not used in safety mode. This explains why they could not be detected, and at the same time confirms that their non-coverage is not a safety issue. All faults that had no effect in the performance mode but turned out to have an effect in safety mode were also detected.

Obviously the poor coverage in performance mode is due to the lack of the comparison with the checker; errors were only detected by the watchdog. In these cases the instruction address of the reconfigurable dual-core system was not changing any more and without the watchdog the system would not switch back to the safety mode as intended. This demonstrates the central role of the watchdog for preventing our system from being hung up in performance mode.

With reference to the aims of our experiments we can conclude, that the proposed reconfigurable dual-core system can detect any single fault – even in the non-duplicated dual-core frame – when operating in the safety mode. Our exhaustive experiments did not reveal any single point of failure. Error detection coverage in performance mode, however, is as poor as that of every non-duplicated system without specific protection. Even though our experiments have shown that all relevant faults are reliably detected upon switching to safety mode, two important points have to be considered: (1) A watchdog is necessary to enforce the switch to safety mode in case the system gets hung up in performance mode. (2) To avoid data pollution in external components it is necessary to protect safety relevant external information. This may be achieved through a write protection for critical memory areas during performance mode.

VII. CONCLUSION

Considering the fact that not every task in an embedded system is safety-critical we have proposed an architecture that can either be configured in a fail safe master/checker fashion for safety critical tasks (safety mode), or in a dual processor fashion for enhanced performance (performance mode). In this paper we have focused on the safety mode. We have shown that with a careful implementation mainly based on the self-checking principle and dual-rail coding complete coverage of single faults can be obtained even for the non-duplicated function units of the reconfigurable dual-core framework. Our analysis has pointed out that in the proposed implementation all single points of failure and all common-mode failures – even at the input, output or error signals – can be eliminated in the safety mode, if several layout rules are obeyed and if the operation of the checker is delayed by 1.5 clock cycles. We have further shown that in safety mode the system does not produce wrong results even in case of error propagation from performance mode. The only requirements for this are the availability of some basic kind of memory protection and the provision of a watchdog. Comprehensive experimental results have confirmed our theoretical analysis: No coverage violation has been observed in safety mode.

We have carefully minimized the assumptions on the processor core and have consequently treated the core as a black box. This makes our solution generic and allows its use in conjunction with virtually any standard processor core. The area overhead of the presented reconfigurable dual core architecture relative to a standard master-checker solution [11] is as low as 11% (using SPEAR cores in both cases).

TABLE II
RESULTS OF THE FAULT INJECTION EXPERIMENTS IN PERFORMANCE MODE

effect of the fault		component			
		master	slave	frame	overall
detected in performance mode	detected without effect	0	0	1473	1473
	detected before effect	0	0	1149	1149
	detected during or after effect	0	0	423	423
	not jumped to lock mode	17203	7713	701	25617
effect but not detected in performance mode, then in safety mode	detected without effect	29475	4492	616	34583
	detected before effect	0	0	0	0
	detected during or after effect	0	0	0	0
	not detected without effect	265	11	182	458
	not detected with effect	0	0	0	0
no effect and not detected in performance mode, in safety mode	detected without effect	64	44066	3585	47715
	detected before effect	0	0	0	0
	detected during or after effect	9542	0	112	9654
	not detected without effect	7517	7782	3261	18560
	not detected with effect	0	0	0	0
		64066	64064	11502	139632

Future work will be directed towards investigating the performance of the architecture in performance mode. Our experiments have confirmed so far that the switching between the modes can indeed be done very fast during runtime. Further efforts will be devoted to developing a strategy for fast recovery of the dual-core in safety mode.

REFERENCES

[1] R. Baumann, *The Impact of Technology Scaling on Soft Error Rate Performance and Limits to the Efficacy of Error Correction*, Electron Devices Meeting 2002. IEDM'02. Digest.International, pp. 329-332, 2002.

[2] G. Georgakos, *Radiation Induced Soft Error Rate for SoC Designs*, Infineon Customer Information, Vers. 2.1, Febr. 2003.

[3] A. Allan, D. Edenfeld, W.H. Joyner, A.B. Kahng, M. Rodgers and Y. Zorian, *2001 Technology Roadmap for Semiconductors*, Computer, Vol. 35, no. 1, pp. 42-53, Jan. 2002.

[4] E. Boehl, T. Lindenkreuz and R. Stephan, *The fail-stop controller AE11*, Proceedings of the International Test Conference, pp. 567-577, 1997.

[5] I.D. Elliott and I.L. Sayers, *Implementation of a 32-bit RISC processor incorporating hardware concurrent error detection and correction*, IEE Proceedings of Computers and Digital Techniques, Vol. 137, No. 1, pp. 88-102, 1990.

[6] R. Russell and I.D. Elliott, *Design of highly reliable VLSI processors incorporating concurrent error detection/correction*, Euro ASIC, pp. 316-321, 1991.

[7] M. Pflanz and H.T. Vierhaus, *Online check and recovery techniques for dependable embedded processors*, Micro, IEEE, Vol. 21, No.5, pp. 24-40, 2001.

[8] J.H. Patel and L.Y. Fung, *Concurrent Error Detection in ALUs by Recomputing with Shifted Operands*, IEEE Transaction on Computers, Vol. C32, No. 7, pp. 589-595, 1982.

[9] J. Li and E.E. Swartzlander, *Concurrent error detection in ALUs by recomputing with rotated operands*, Defect and Fault Tolerance in VLSI Systems, Proceedings of the IEEE International Workshop on, pp. 109-116, 1992.

[10] A. Steininger and C. Scherrer, *Identifying Efficient Combinations of Error Detection Mechanisms Based on Results of Fault Injection Experiments*,IEEE Transactions on Computers, Vol. 51, No.2, pp. 235-239, 2002.

[11] T. Kottke and A. Steininger, *A Generic Dual Core Architecture*, Proceedings of the IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop, pp. 159-166, 2004.

[12] R. Angerbauer, E. Boehl, Y.von Collani, B. Fehrenbacher, R. Gmehlich, C. Graebitz, W. Harter, F. Hartwich, T. Kottke, J. Lutz, B. Mueller, W. Pfeiffer and R. Weiberle, Bosch internal report.

[13] *MC88100 32-Bit RISC Microprocessor Technical Summary*, Document MC88100/D, Mototola Inc. 1990.

[14] D.E. Lenoski, *A highly integrated, fault-tolerant minicomputer: the NonStop CLX*, Compcon Spring 88, Thirty-Third IEEE Computer Society International Conference, Digest of Papers, pp. 514-519, 1988.

[15] R. Kalla, B. Sinharoy and J.M. Tendler, *IBM Power5 chip: a dual-core multithread processor*, Micro, IEEE, Vol. 24, No. 2, pp. 40-47, 2004.

[16] T. Takayanagi, J.L. Shin, J. Su, A.S. Leon, *Deep-submicron design challenges for a dual-core 64b UltraSPARC microprocessor implementation*, International Conference on Integrated Circuit Design and Technology, pp. 147-150, 2004.

[17] O. Gonzalez, H. Shrikumar, J.A. Stankovic and K. Ramamritham, *Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling*, Proceedings of the 18th IEEE Real-Time Systems Symposium, pp. 79-89, 1997.

[18] M. Delvai, *SPEAR Handbook*, Technical Report, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[19] M. Delvai, W. Huber, P. Puschner and A. Steininger, *Processor Support for Temporal Predictability - The SPEAR Design Example*, Proceedings of the 15th Euromicro Conference on Real-Time Systems, pp. 169-176, 2003.

[20] T. Kottke and A. Steininger, *A Fail Silent Memory for Automotive Applications*, IEEE European Test Symposium, Infomal Digest of Papers, pp. 253-258, 2004.

[21] S.R. Manthani and S.M. Reddy, *On CMOS Totally Self-Checking Circuits*, Proceedings of the International Test Conference, pp. 866-877, 1984.

[22] M. Abramovici, M.A. Breuer and A.D. Friedman, *Digital Systems Testing & Testable Design*, Wiley-IEEE Press, 1994.

[23] S. Kundu, E.S. Sogomonyan, M. Goessel and S. Tarnick, *Self-checking comparator with one periodic output*, IEEE Transactions on Computers, Vol. 45, No. 3, pp. 379-380, 1996.

[24] N. Kanekawa, T. Meguro, K. Isono, Y. Shima, N. Miyazaki and S. Yamaguchi, *Fault Detection and Recovery Coverage Improvement by Clock Synchronized Duplicated Systems with Optional Time Diversity*, IEEE Proceedings of FTCS, Vol. 28, pp. 196-200, 1998.

[25] A.M. Usas, *A Totally Self-Checking Checker Design for the Detection of Errors in Periodic Signals*, IEEE Transactions on Computers, Vol. C-24, No. 5, pp. 483-489, 1975.

[26] T. Kottke and A. Steininger, *Dual Core Architecture with Error Containment*, East-West Design & Test International Workshop, Proceedings of, pp. 102-108, 2004.

[27] W.C. Carter and P.R. Schneider, *Design of Dynamically Checked Computers*, Proc. IFIP'68 World Computer Congress, Amsterdam, The Netherlands, pp. 878-883, 1968.

[28] W.C. Carter, *Hardware Fault Tolerance*, Resilient Computer Systems (edited by T. Anderson), Collins, London, pp. 11-63, 1985.