A Flexible Scheme for Scheduling Fault-Tolerant Real-Time Tasks on Multiprocessors

Michele Cirinei[1], Enrico Bini[1], Giuseppe Lipari[1], Alberto Ferrari[2]

[1]Scuola Superiore Sant'Anna
56127 Pisa, Italy
{m.cirinei,e.bini,lipari}@sssup.it

[2]PARADES EEIG
00186 Roma, Italy
aferrari@parades.cnr.rm.it

Abstract

The recent introduction of multicore system-on-a-chip architectures for embedded systems opens a new range of possibilities for both increasing the processing power and improving the fault-robustness of real-time embedded applications. Fault-tolerance and performance are often contrasting requirements. Techniques to improve robustness to hardware faults are based on replication of hardware and/or software. Conversely, techniques to improve performance are based on exploiting inherent parallelism of multiprocessor architectures.

In this paper, we propose a technique that allows the user to trade-off parallelism with fault-tolerance in a multicore hardware architecture. Our technique is based on a combination of hardware mechanisms and real-time operating system mechanisms. In particular, we apply hierarchical scheduling techniques to efficiently support fault-tolerant, fault-silent and non-fault-tolerant tasks in the same system.

1. Introduction

Recently, there has been considerable interest in using multicore system-on-a-chip architectures in the embedded systems domain. This interest is mainly due to the need for increasing the computational power maintaining low both the clock frequency and the complexity of each core. Moreover, trends in technology scaling allow the increase of both the number and the power of the processors that can be integrated in a single chip, making multicore systems-on-a-chip even more appealing for embedded systems.

However, it is well-known [7, 20] that technology scaling sensitizes electronic devices to external disturbs. Reasons of this fact relate to several aspects of scaling, such as lower voltage levels, lower capacitances, higher working frequency. The overall effect is an higher probability that lower energy particles (such as alpha particles) cause temporary bit-flipping in memory and logic circuits (so called *soft errors*). For this reason, tolerance to soft errors is bound to become a major aspect in system design.

The classical way of providing fault-tolerance on multicore platforms (which are generalized by multiprocessors) is to use time and/or space redundancy. In time redundancy, the same software is executed two or more times on the same CPU, and the produced results are compared. However, final results could be influenced by the effective instants of execution, possibly leading to several different, although correct, results. In such a case, comparison is difficult. In space redundancy, on the contrary, the same software is executed at the same time on different CPUs. One solution based on this idea is to execute replicas in *lock-step*: each involved processor executes the *same* code at the *same* time (i.e., step by step), and every result is compared, instantly revealing differences in operands or result of each single instruction. To simplify the circuitry one idea is to reduce the comparison points, for example considering only I/O operations. This way, we also obtain that comparisons can be completely implemented outside the processor (e.g., at the interconnection between processor and bus).

By using these techniques, it is possible to implement a *fault-tolerant* system (with more than 2 CPUs and by using an appropriate voting or fault-masking mechanism), or a *fail-silent* system (with two CPUs and a comparator) where the fault is simply detected but not corrected. In both cases, the fault-tolerant behaviour is achieved at the cost of a reduced computational power.

This approach is usually static, in the sense that the configuration does not vary in time. Hence the fault-tolerance characteristics and the performance are not adjusted on the application requirements. However, such a limitation may be too restrictive because not all software tasks are fault-critical. In general, applications consist of a mixture of fault-tolerant, fail-silent and non-fault-tolerant tasks. It

1-4244-0910-1/07/$20.00 ©2007 IEEE.

would be desirable to use the multiprocessor platform at its best: as a replicated hardware platform for fault-tolerant and fail-silent tasks; and as a parallel processor platform for the non-fault-tolerant tasks. Unfortunately, in classical fault-tolerant systems, such a degree of flexibility is not possible.

Contributions In this paper we propose a technique, based on dynamic on-line reconfiguration of a four-processor multicore hardware platform, to achieve a trade-off between performance (through parallelism) and fault-tolerance (through replication). Our technique consists in dividing the time-line into time slots, each one dedicated to a different class of tasks: fault-tolerant, fail-silent, and non-fault-tolerant tasks. At each slot, we dynamically reconfigure the hardware platform to support a certain degree of replication or parallelism. To guarantee the schedulability of each class of tasks in its slot, we apply the theory of hierarchical scheduling [15, 19, 12].

Based on that, we propose a technique to configure the platform to achieve different goals. We then propose two examples: first we show how to tune the platform to minimize the processor bandwidth wasted in overhead; then we consider how to maximize its flexibility at run-time.

1.1. Related works

The problem of fault-tolerance in multiprocessors systems has been thoroughly addressed. Two main branches are usually proposed: on one side, try to build stronger hardware platforms, able to resist to faults and continue their work at the same or at a degraded level; on the other side, to make the application able to recover from faults, either by time or space redundancy.

The literature about hardware fault-tolerance techniques is extremely vast, and an exhaustive analysis is almost impossible. To protect the application from faults, a general approach is *redundancy*. The idea is to introduce redundant copies of the elements to be protected (processors or other components), and exploit them in the event of a fault.

We focus our research on the so called *lock-step configuration*, in which the redundancy is used for both fault detection and recovery. In a lock-step configuration (explained in more detail in Section 2.4), two or more processors execute the same code cycle by cycle, and a dedicated circuitry compares the results. When a fault is detected, recovery can be performed via SW (e.g., checkpoints and re-execution) and/or HW (e.g., reconfiguration of the application on the remaining processors).

Far from being only a theoretical fault-tolerance technique, the lock-step approach has been applied in several commercial systems. The core of the Sequoia computer [8] was an high number of *Processor Elements*, each one composed of two Motorola MC68020 operating in lock-step and a comparator testing the identicalness of the results. In more recent years, the lock-step idea is exploited, e.g., as part of the Continuous Processing Technology implemented in the Stratus ftServer family of products. Similarly, in the HP NonStop Advanced Architecture it is possible to configure two Intel Itanium processors to work in lock-step [1].

Similar solutions exist for embedded systems. Xilinx produces the Virtex-II Pro FPGA, based on two IBM PowerPC 405, and uses the FPGA as the core of its ML310 Embedded Development Platform. One of the proposed applications for the ML310 Platform is a processor lock-step for fault-tolerant applications [3]. The IBM PowerPC 750GX processor includes a *lock-step facility* [2], which integrates all the circuitry, from comparator to data steering, necessary for the connection of two identical processors in lock-step.

A major disadvantage in the way this technique is implemented in commercial systems is the lack of flexibility: despite the fact that every system is composed of tasks with different requirements of reliability, the fault-tolerance techniques do not vary in time. In this sense, the architecture we propose is extremely more flexible, since its fault-tolerance techniques can be tuned on the specific application, limiting the necessary loss in computational power.

In the field of software fault-tolerance techniques, scheduling algorithms with safety characteristics are particularly interesting. A long series of works from the Real-Time Systems Research Group at University of York, and summarized in [18, 14], propose different solutions for the schedulability of real-time task sets under fixed priority, considering various assumptions on faults and fault-tolerance techniques. Another interesting approach is based on the well-known concept of primary/backup [11, 17], in which, for each critical task in the system, a backup copy is activated when a fault impairs the primary one.

2. System model

2.1. Fault model

In this paper we address soft errors in multicore systems. As explained earlier, soft errors are transient errors in memory or logic due to alpha particles, neutrons or similar low energy particles. Due to the nature of soft errors, we consider each fault to be transient, which means that the faulty condition lasts for a limited and short time interval, after which traces of the fault remain only in possible wrong values. Moreover, since soft error rates statistically guarantee that time between two failures is sufficient to perform simple recovery operations, we assume that only one fault can affect the system at a time. This allows us to rely on the *single transient fault assumption*.

In a multicore environment all cores are integrated in the same chip. Hence one could think that a single faulty event could bring to simultaneous or correlated errors in different processors. Under the hypothesis that only soft errors influence the system, this problem does not show up, since a single particle can strike only one core. This means that even in a multicore environment the single transient fault assumption does make sense.

It is outside the scope of this paper to discuss how a fault can be recovered. This will be the topic of future research. Informally, we can say that since the fault is transient, the recovery involves three steps: waiting for the end of the fault; correcting data errors due to tasks aborted in an inconsistent state or wrong results written by non protected tasks; restarting some of the non protected tasks that were influenced by the fault.

2.2. Operating modes

An environment prone to faults must be protected to avoid potentially catastrophic situations. It is clear, though, that not all the parts of the application have the same importance, and different aspects can require different levels of fault-robustness. Consider an application which controls a car engine and shows its activity on a screen. While we could accept the visualization to be degraded, the control algorithm must produce the correct result despite the presence of faults. This idea is expressed by the concept of *operating mode* (or *mode*, for short). Intuitively, different parts of the application require to execute in distinct modes, every one characterized by a specific degree of tolerance to faults. In particular, under the single transient fault assumption, the following 3 modes can be required: in **fault-tolerant mode** (FT) the system is not damaged by the presence of a fault, i.e. wrong results can never be produced; in **fail-silent mode** (FS), in case of a fault the system (or the faulty part of it) is made silent, in order to avoid errors and wrong results to propagate; in **non-fault-tolerant mode** (NF) no fault-tolerance guarantees are given, i.e. the behaviour of the system in the presence of a fault is unpredictable.

2.3. Application model

An application consists of a set of independent real-time tasks to be executed on a multiprocessor platform. A real-time task τ_i is characterized by the triplet (C_i, T_i, D_i), where C_i denotes the worst-case computation time, T_i represents the minimum interarrival time (i.e. the time separation between two consecutive activations), and D_i is the relative deadline with $D_i \leq T_i$. This model corresponds to the sporadic task model. An important characteristic of the task τ_i is also its utilization U_i which is set equal to $\frac{C_i}{T_i}$. For any given subset of tasks $\mathcal{T}$ we denote its utilization by $U(\mathcal{T})$ and we naturally set it equal to $\sum_{\tau_i \in \mathcal{T}} U_i$. Finally, it is important to point out that we consider all tasks to be independent, i.e. they do not share data with critical sections.

Depending on its desired robustness to faults, every task requires to execute in one of the modes described in Section 2.2, and is defined to be a fault-tolerant (FT), fail-silent (FS) or non-fault-tolerant (NF) task. Based on the required mode represented by parameter $mode_i$, tasks are then partitioned into three sets: $\mathcal{T}_{\mathsf{FT}}$ for FT tasks, $\mathcal{T}_{\mathsf{FS}}$ for FS tasks, and $\mathcal{T}_{\mathsf{NF}}$ NF tasks. We assume that the task set is fixed and known before run-time, i.e. no task is dynamically created or deleted from the system.

Given an hardware platform able to provide the modes described above, the goal of this work is to schedule the application such that for every task no deadline is missed and the required mode is guaranteed.

2.4. Hardware architecture

The hardware architecture is based on the concept of *lock-step*. The generic lock-step configuration is composed of two identical processors and a monitor. The two processors execute the same instruction cycle by cycle, so that in a fault free situation their local context remains synchronized. Meanwhile, the monitor compares all the outputs from the processors: if the outputs are the same, it is assumed that there are no faults, so the access to the bus is granted, and memory is read/written; otherwise, the access is blocked and an error signal is raised. This architecture guarantees that a processor error is intercepted before it can propagate to the main memory (or to the input/output subsystem), hence the memory integrity is preserved. Of course, it is possible to force more than two processors to execute the same code, building the so called *redundant lock-step*, and obtaining a more robust channel.

Exploiting the lock-step concept, and slightly modifying the Shared Memory Dual Lock-Step Architecture proposed by Baleani et al. [5], we consider an hardware platform composed of 4 processors and a shared memory architecture, depicted in Figure 1.

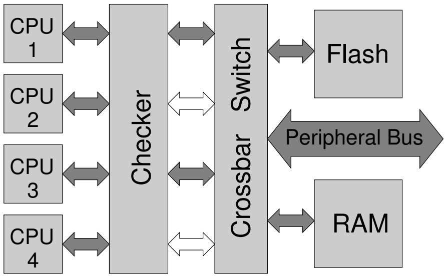

Figure 1. The hardware platform

The key element introduced in our platform is the *checker*, which conceptually integrates three aspects: the comparison of the results provided by the processors, the control of the bus and memory access, and the reconfiguration of the platform.

The checker can change on-line its internal configuration, in order to provide to the application the three operating modes described in Section 2.2. At each instant, the 4 processors can work in one of the following modes:

- all 4 in redundant lock-step, to provide the FT mode: due to the single fault assumption, only one processor can fail, so the correct result can be guessed by majority. The 4 processors build a single fault-tolerant channel;

- coupled in 2 lock-step, providing the FS mode: again only one processor in a couple can fail, so a mismatch between the two outputs is immediatly revealed and the channel is blocked. The two couples realize two independent fail-silent channels;
- in parallel, implementing the NF mode: the 4 processors work independently, and no fault-tolerance guarantees are given, whereas the highest computational power is delivered.

The modes are provided to the application by periodically switching from a mode to another, maintaining a sort of temporal separation between different modes. We call *mode switch* the on-line change of the system configuration between two modes. So, in every time interval, one of the modes is selected, and only tasks requiring that mode can execute. How the reconfiguration of the platform is actually performed is out of the scope of this paper.

In practice, the time line is divided into intervals of length P, each one composed of three slots of length Q_{FT}, Q_{FS} and Q_{NF}, one for each mode respectively. Accordingly, the subset $\mathcal{T}_{\mathsf{FT}}$ will be executed during the first slot of length Q_{FT}, the subset $\mathcal{T}_{\mathsf{FS}}$ during the second slot of length Q_{FS}, and the subset $\mathcal{T}_{\mathsf{NF}}$ during the last slot of length Q_{NF}.

Another aspect we have to consider is that the mode switch requires some operations, such as task state synchronization and data storing. Hence O_{FT}, O_{FS} and O_{NF} represent respectively the overheads when switching out of modes FT, FS and NF. Moreover, we define the sum of the three overheads $O_{\text{tot}} = O_{\mathsf{FT}} + O_{\mathsf{FS}} + O_{\mathsf{NF}}$. Note that the overhead consumes part of the time available to the respective subset, so the generic overhead O_k is included in the slot Q_k. This leads to the definition of $\tilde{Q}_k = Q_k - O_k$ as the amount of time available to the tasks in the generic mode k. The notation is depicted in Figure 2.

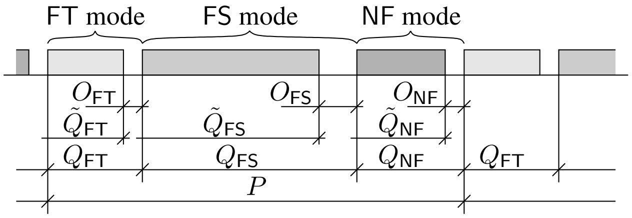

Figure 2. Switching between modes

We point out that in FT mode 3 processors are sufficient to provide a fault-tolerant channel. Thus, it might be possible to use the fourth processor for running NF tasks. However, this additional degree of flexibility would complicate both the hardware management circuitry and the operating system. Thus, to simplify the platform management, we prefer to maintain the modes separated in time. Please note that from this point of view, there is no difference between considering all the 4 processors together in a fault-tolerant channel, and using only 3 processors for the fault-tolerant channel and shutting down the 4th processor.

Note that in order to provide any level of fault-robustness to the application, it is necessary to protect the whole platform from faults, including memory, bus, interrupt controllers, etc. In this paper we only focus on the tolerance from faults on the processors. For the rest of the platform we consider to apply other well-known hardware fault-tolerance techniques, such as ECCs for bus and memory.

3. Design methodology

The proposed architecture opens a new wide range of possibilities for exploiting the trade-off between fault-tolerance, implemented by hardware replication, and computational power, provided by parallel execution. When designing the fault-tolerant platform, we already know the set of tasks to be executed and their desired robustness to faults. The final design must guarantee that each task completes within the assigned deadline, and executes in the required mode (NF, FS or FT)

When the platform provides some degree of parallelism (i.e. during modes FS and NF) it is necessary to decide how to schedule the tasks on the multiprocessor. Two main classes of algorithms are known in the literature: the *partitioning* [16, 6] and the *global* [21] strategies. For lack of space, we do not discuss advantages and disadvantages of each class of algorithms; in this paper we focus on the partitioned scheme (in which each task is statically assigned to one processor), whereas the analysis of global strategies is postponed to future works. Moreover, since the goal of this paper is the tuning of the platform once the characteristics of the application are known, we consider the tasks to be manually partitioned (although suitable solutions to automatically partition the tasks among the processors exist [6]).

During NF mode, four processors are available. Hence the tasks in $\mathcal{T}_{\mathsf{NF}}$ are partitioned into the 4 subsets $\mathcal{T}^1_{\mathsf{NF}}$, $\mathcal{T}^2_{\mathsf{NF}}$, $\mathcal{T}^3_{\mathsf{NF}}$ and $\mathcal{T}^4_{\mathsf{NF}}$, depending on the processor they will run on. Similarly the tasks in $\mathcal{T}_{\mathsf{FS}}$ are partitioned into the 2 subsets $\mathcal{T}^1_{\mathsf{FS}}$ and $\mathcal{T}^2_{\mathsf{FS}}$. Finally, during FT mode only one processor is available hence no further partition is required.

Once tasks are partitioned, the schedulability problem leads to the well studied case of one single processor. However each subset of tasks can only run during the dedicated mode, which is allocated only a fraction of the total available time. A considerable amount of efforts have been recently dedicated to the study of such a problem. This scheduling problem is generally refered to as *hierarchical scheduling*, and it is recalled in Section 3.1.

Finally, let us formally define the problem: given a set of real-time tasks $\tau_i = (C_i, T_i, D_i, mode_i)$ defined as in Section 2.3, and a 4-processors hardware platform as described in Section 2.4, and assuming that the switching overhead for mode k is O_k, and a partitioning scheduling strategy is adopted, what are the parameters (P and Q_k for each mode k) of the operational modes that guarantee the schedulability of each task τ_i in the required mode $mode_i$?

3.1. Hierarchical Scheduling

As shown in Figure 2, $\tilde{Q}_{\mathsf{FT}}$, $\tilde{Q}_{\mathsf{FS}}$, and $\tilde{Q}_{\mathsf{NF}}$ represent respectively the amount of time actually available to $\mathcal{T}_{\mathsf{FT}}$, $\mathcal{T}_{\mathsf{FS}}$, and $\mathcal{T}_{\mathsf{NF}}$ tasks.

We remark that the tolerance to faults of the system is higher when $\tilde{Q}_{\mathsf{FT}}$ dominates the other values. On the other hand, greater values of $\tilde{Q}_{\mathsf{NF}}$ maximize the delivered computational power, since four processors are available in NF mode.

We can estimate the *computational power* provided during each mode by a supply function, which is defined as follows.

Definition 1 *Given a mode $k \in \{\mathsf{FT}, \mathsf{FS}, \mathsf{NF}\}$, the supply function $Z_k(t)$ of the mode k is the* minimum amount of time *provided during the mode k in any interval whose length is t. Formally,*

$$Z_k(t) = \min_{t_0}\{\textit{time provided in } [t_0, t_0 + t] \textit{ during mode } k\}.$$

The introduction of the supply function is very useful for verifying the schedulability of real-time tasks, because it provides a minimum time guarantee which is granted under any circumstances. The idea of the supply function has been already exploited both in the field of real-time [15, 19, 4] and in networking [13].

In Figure 3 we show the supply function of each mode (please refer to [15, 19, 4, 13] for a detailed explaination).

Lemma 1 (from [15]) *The supply function of a mode $k \in \{\mathsf{FT}, \mathsf{FS}, \mathsf{NF}\}$ is the following*

$$Z_k(t) = \begin{cases} j\,\tilde{Q}_k & \textit{if } t \in [j\,P, (j+1)\,P - \tilde{Q}_k) \\ t - (j+1)(P - \tilde{Q}_k) & \textit{otherwise} \end{cases} \tag{1}$$

where $j = \lfloor \frac{t}{P} \rfloor$.

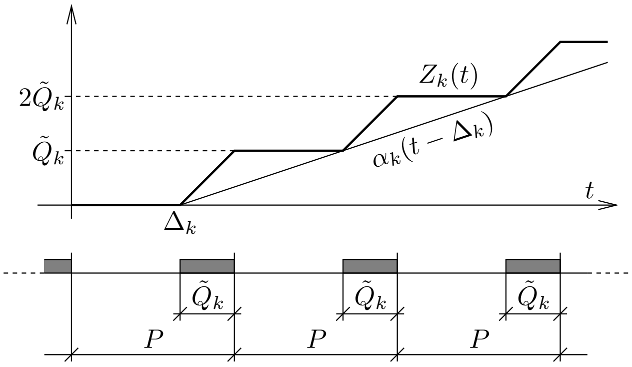

Figure 3. The supply function

From the supply function $Z_k(t)$ of each mode, we derive two important features: the *rate* α_k, which roughly denotes the fraction of processor available at mode k, and the *delay* Δ_k, which is the maximum amount of time a task executing during mode k must wait without being serviced.

It is possible to prove [15] that the relationship between (α_k, Δ_k) and the mode parameters is:

$$\alpha_k = \frac{\tilde{Q}_k}{P} \qquad \Delta_k = P - \tilde{Q}_k \tag{2}$$

The values of α_k and Δ_k are useful, since they introduce a simple lower bound of the supply function, as follows

$$Z'_k(t) = \max\{0, \alpha_k(t - \Delta_k)\}, \tag{3}$$

as it can be noticed in Figure 3. Since we always have $Z'(t) \leq Z(t)$, assuming $Z'(t)$ as supply function is safe, meaning that every solution feasible with $Z'(t)$ is always feasible with $Z(t)$. However, for simplicity, we consider the supply function equal to $Z'(t)$, and from now on $Z(t)$ will denote the supply function of Eq. (3). The full consideration of the exact $Z(t)$ does not present any conceptual difficulty, but it is only tedious to develop the math properly.

3.2. Schedulability analysis

While the classical approach is based on verifying the schedulability of a task set once the supply function is given, we focus on the opposite problem: given a task set, what are all the possible supply functions which guarantee the task deadlines?

FP scheduler If tasks are scheduled by fixed priorities the feasibility condition of a task set $\mathcal{T}$ allocated to a mode k, characterized by a rate α_k and a delay Δ_k is provided by the following theorem.

Theorem 1 (Theorem 3 in [15]) *A task set $\mathcal{T}$ is schedulable by Fixed Priorities (FP) in the mode k, described by (α_k, Δ_k), if:*

$$\forall \tau_i \in \mathcal{T} \quad \exists\, t \in \mathsf{schedP}_i \quad \Delta_k \leq t - \frac{W_i(t)}{\alpha_k} \tag{4}$$

where

$$W_i(t) = C_i + \sum_{\tau_j \in \mathsf{hp}(\mathcal{T}, \tau_i)} \left\lceil \frac{t}{T_j} \right\rceil C_j \tag{5}$$

and (i) schedP_i *is the set of* scheduling points *of task τ_i as defined in [10] and (ii) $\mathsf{hp}(\mathcal{T}, \tau_i)$ is the set of tasks in $\mathcal{T}$ with a higher priority than τ_i.*

The set of scheduling points schedP_i is the smallest set of points where Eq. (4) must be checked for the task τ_i to be schedulable (please refer to [10], for further details).

Substituting in Eq. (4), the expressions of α_k and Δ_k from Eq. (2), and performing a sequence of algebraic manipulations we find an explicit relationship between the lengths of the time slots $\tilde{Q}_k$ and the period P:

$$\tilde{Q}_k \geq \max_{\tau_i \in \mathcal{T}} \min_{t \in \mathsf{schedP}_i} \frac{\sqrt{(t-P)^2 + 4\,P\,W_i(t)} - (t-P)}{2} \tag{6}$$

Notice that the right hand side of Eq. (6) depends on the period P, the task set $\mathcal{T}$, and the adopted scheduling policy, which is fixed priorities. We can then compact Eq. (6) in a very convenient way as follows

$$\tilde{Q}_k \geq \mathsf{minQ}(\mathcal{T}, \mathsf{FP}, P) \tag{7}$$

where all the complex dependecies of Eq. (6) are hidden in the function minQ.

EDF scheduler If tasks are scheduled by EDF then we can use the results of hierarchical scheduling when the local scheduler is EDF [19, 9]. If the supply function has a rate α_k and a delay Δ_k, then Theorem 2 ensure the feasiblity of an EDF task set.

Theorem 2 (from [9]) *A task set $\mathcal{T}$ is schedulable by Earliest Deadline First (EDF) in the mode k, described by (α_k, Δ_k), if:*

$$\forall t \in \mathsf{dlSet}(\mathcal{T}) \quad \Delta_k \leq t - \frac{W(t)}{\alpha_k} \tag{8}$$

where

$$W(t) = \sum_{i=1}^{n} \max\left\{ \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor, 0 \right\} C_i \tag{9}$$

and $\mathsf{dlSet}(\mathcal{T})$ is the set of all deadlines up to the hyperperiod of the tasks in $\mathcal{T}$.

Notice that the previous formulation also applies to task set with static offset and jitter. However, we develop all our results in the simpler case of no offset/jitter, because the full treatment does not present theoretical problem but the math is heavier (see [9] for further details).

Using the same arguments which led to Eq. (7), we can equivalently find the conditions on the time quantum $\tilde{Q}_k$ such that the mode k can feasibly schedule all the tasks allocated to it if they are scheduled by EDF. In fact we have

$$\tilde{Q}_k \geq \mathsf{minQ}(\mathcal{T}, \mathsf{EDF}, P) \tag{10}$$

where in this case

$$\mathsf{minQ}(\mathcal{T}, \mathsf{EDF}, P) = \max_{t \in \mathsf{dlSet}(\mathcal{T})} \frac{\sqrt{(t-P)^2 + 4\,P\,W(t)} - (t-P)}{2} \tag{11}$$

3.3. Integration between modes

Until now, we have only considered generic task sets $\mathcal{T}$ in isolation and the relationship between the time quanta Q_k's is neglected. Now we want to combine the equations which describe the feasible time quanta to find all the feasible $(Q_{\mathsf{NF}}, Q_{\mathsf{FS}}, Q_{\mathsf{FT}})$. Let us generically denote by alg the scheduling algorithm adopted to schedule the tasks.

During FT mode, the quantum Q_{FT} must be large enough to schedule within the deadlines all the tasks in $\mathcal{T}_{\mathsf{FT}}$. Recalling that $\tilde{Q}_{\mathsf{FT}} = Q_{\mathsf{FT}} - O_{\mathsf{FT}}$, the condition on Q_{FT} is

$$Q_{\mathsf{FT}} - \mathsf{minQ}(\mathcal{T}_{\mathsf{FT}}, \mathsf{alg}, P) \geq O_{\mathsf{FT}} \tag{12}$$

During FS mode the computational resource equivalent to two processors is available. Hence the time quantum Q_{FS} must accomodate the tasks of both the sets $\mathcal{T}_{\mathsf{FS}}^1$ and $\mathcal{T}_{\mathsf{FS}}^2$. The feasibility of both the sets is ensured by selecting Q_{FS} as follows

$$Q_{\mathsf{FS}} - \max_{i \in \{1,2\}} \mathsf{minQ}(\mathcal{T}_{\mathsf{FS}}^i, \mathsf{alg}, P) \geq O_{\mathsf{FS}} \tag{13}$$

By doing so, in fact, the allocated quantum satisfy the feasibility condition of $\mathcal{T}_{\mathsf{FS}}^1$ and $\mathcal{T}_{\mathsf{FS}}^2$.

Finally, following the same previous arguments, the tasks allocated in NF mode do not miss any deadline if the following condition is verified

$$Q_{\mathsf{NF}} - \max_{i \in \{1,2,3,4\}} \mathsf{minQ}(\mathcal{T}_{\mathsf{NF}}^i, \mathsf{alg}, P) \geq O_{\mathsf{NF}} \tag{14}$$

All the three Equations (12), (13) and (14) define a lower bound to the admissible values for the time quanta as a function of the period P.

The inequalities of Eq. (12), (13) and (14) can be very conveniently summed side by side, revealing the following interesting condition on the period P:

$$P - \sum_{k \in \{\mathsf{FT},\mathsf{FS},\mathsf{NF}\}} \max_{i=1,\ldots,\mathsf{numP}_k} \mathsf{minQ}(\mathcal{T}_k^i, \mathsf{alg}, P) \geq O_{\mathrm{tot}} \tag{15}$$

Note that a value of P fullfilling Eq. (15) does not necessarily determine a feasible solution, unless the lengths of the time quanta satisfy the three Equations (12), (13), and (14). The relationships must be considered as a set of instruments that support the designer during the selection of the parameters. The final choice depends on the specific goal the designer wants to achieve.

In order to clarify the application of the proposed technique, in the next section we show how two different design goals bring to two different solutions.

4. Example of Application

We consider an application composed of 13 real-time tasks requiring different operating modes. Table 1 reports the operating mode required by each task, the task index, the computation times, and the periods. The deadlines are assumed equal to the period for simplicity, although our proposed method well applies also to the case when $D_i \leq T_i$.

Mode	NF					FS				FT			
i	1	2	3	4	5	6	7	8	9	10	11	12	13
C_i	1	1	1	2	6	1	1	2	1	1	1	1	2
T_i	6	8	12	10	24	10	15	20	4	12	15	20	30

Table 1. The task set data

NF tasks are partitioned to the four processors available in NF mode as follows $\mathcal{T}_{\mathsf{NF}}^1 = \{\tau_1\}$, $\mathcal{T}_{\mathsf{NF}}^2 = \{\tau_2, \tau_3\}$, $\mathcal{T}_{\mathsf{NF}}^3 = \{\tau_4\}$ and $\mathcal{T}_{\mathsf{NF}}^4 = \{\tau_5\}$. Similarly, FS tasks need to be partitioned in two groups. The two subsets are $\mathcal{T}_{\mathsf{FS}}^1 =$

$\{\tau_6, \tau_7, \tau_8\}$ and $\mathcal{T}^2_{\mathsf{FS}} = \{\tau_9\}$. All FT tasks run on a unique fault-tolerant processor and are not partitioned.

Once the tasks are partitioned on processors, a scheduling algorithm must be selected. In this example we explore two possible solutions: using RM to schedule the tasks in all the subsets, and using EDF.

After the task partitioning and a suitable selection of the scheduling algorithm, Eq. (15) describes all the feasible values of periods P. We developed a simple Matlab routine (available on the Web at `http://feanor.sssup.it/~bini/faultMP/`) to compute all the feasible periods resulting from Eq. (15). Figure 4 shows the region of feasible periods for both EDF and RM. Notice that, as expected, the EDF region is larger than the RM one, because every RM schedulable task set is also schedulable under EDF.

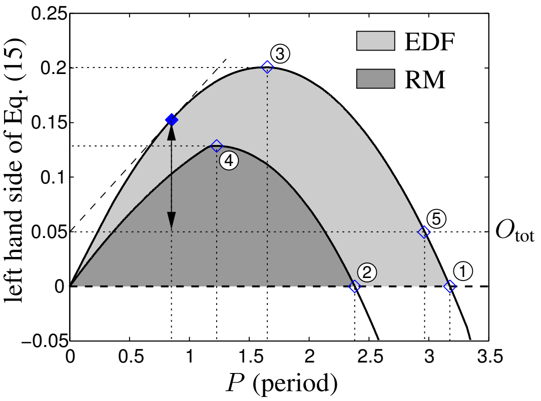

Figure 4. Determining the feasible periods

The feasible regions are above the zero, because $O_{\text{tot}} \geq 0$ and hence, below the zero it wouldn't exist any feasible period P. If the overhead is 0 then the maximum feasible period P is 3.176 if using EDF and 2.381 if RM (points 1 and 2 in the figure). From the figure we can also find the maximum admissible total overahead to have a feasible solution which is 0.201 when using EDF and 0.129 when RM (points 3 and 4).

Let us suppose a realistic example where the total overhead is an intermediate value, e.g. 0.05 (also depicted in Figure 4). A first possible design goal may be to **minimize the bandwidth wasted in overhead $\frac{O_{\text{tot}}}{P}$**. This goal is achieved selecting the maximum feasible period. If the adopted scheduler is EDF, the maximum period is 2.966 (point 5). In correspondence to this period we can now find the admissible values of $\tilde{Q}_{\mathsf{FT}}$, $\tilde{Q}_{\mathsf{FS}}$ and $\tilde{Q}_{\mathsf{NF}}$ from Eq. (12), (13) and (14) respectively (see Table 2(b)).

In order to make the reader confident of the correctness of these results we remind that a **necessary** condition for the schedulability of a task set $\mathcal{T}$ is that the allocated bandwith is not smaller than the task set utilization $U(\mathcal{T})$ (Table 2(a)). Let us verify it for the tasks in $\mathcal{T}_{\mathsf{NF}}$, leaving to the reader the check for the two other modes FT and FS. Since the bandwidth provided by $\tilde{Q}_{\mathsf{NF}}$ must be sufficient to serve every subset of $\mathcal{T}_{\mathsf{NF}}$, we obtain the following inequality:

$$\frac{\tilde{Q}_{\mathsf{NF}}}{P} = 0.275 \geq \max_{i=1,\ldots,4} U(\mathcal{T}^i_{\mathsf{NF}}) = 0.250$$

Since this design choice does minimize the bandwidth wasted in the mode switches, it provides the higher amount of unused bandwidth in each mode, because the time quanta are allocated at their maximum.

By following this design goal, inequalities (12), (13) and (14) must hold with the equal sign and selecting any larger time quantum would violate the last constraint. This phenomenon happens because we have selected a period value on the boundary of the feasible region. This solution, however, may present a critical aspect: if we allow tasks to dinamically arrive and leave the system, we can only take advantage of the unused bandwidth on each mode, but the length of the time quanta cannot be modified at all.

Instead there may be design scenarios where some tasks arrive dynamically and it would be very convenient to shrink or enlarge the time quanta. How do we proceed in this hypothesis? What does the goal of our design become?

Informally speaking, we would like to select the period such that the **time quanta can vary as much as possible at run-time**. Basically we would like to redistribute, if necessary, the most possible bandwidth among the modes. Let us explain graphically how to do it. In Figure 4, the vertical distance from any point (P, O_{tot}) to the boundary measures the slack amount in Eq. (15) , when the period is set equal to P. The slack bandwidth is maximum when the ratio between the slack amount and the period P is maximum (indicated in Figure 4 by the maximum slope of the dashed line). The solution found in this case is reported in Table 2(c). Notice that 12.1% of the bandwidth can be redistributed dynamically, although in this case the allocated bandwidth is much tighter to the required amount.

		P	O_{tot}	$\tilde{Q}_{\mathsf{FT}}$	$\tilde{Q}_{\mathsf{FS}}$	$\tilde{Q}_{\mathsf{NF}}$	slack
(a)	req. util.			0.267	0.267	0.250	
(b)	length	2.966	0.050	0.820	1.281	0.815	0.000
	alloc. util.	1.000	0.017	0.276	0.432	0.275	0.000
(c)	length	0.855	0.050	0.230	0.252	0.220	0.103
	alloc. util.	1.000	0.059	0.269	0.294	0.257	0.121

Table 2. Possible design solutions

As a final remark, please notice that all the computation are developed in the case of an EDF scheduler. The same reasoning applies to the RM scheduling algorithm as well.

5. Conclusions and future work

In this paper, we propose a flexible management scheme of a symmetrical multiprocessor platform for scheduling periodic real-time tasks with fault-robustness requirements.

To the best of the authors' knowledge, this is the first attempt to dynamically reconfigure a hardware architecture in different operating modes (lock-step or parallel) to support at the same time fault-tolerance by means of hardware replication, and parallel execution for high performance. The flexibility of the proposed platform can be used to achieve the best trade-off between hardware replication and parallelism. The methodology takes advantage of an hardware platform capable of exploiting a high parallelism as well as a high replication. We propose a time partitioning scheme to alternate parallel execution modes to lock-step modes. Finally, we propose an algorithm to find the optimal configuration of time partition parameters to minimize the system overhead.

Our algorithm can be seen as a first step towards a complete methodology for designing real-time fault-tolerant systems based on a multiprocessor. As explained in Section 3, the *task allocation problem* needs to be solved. Also, in the future we will explore the possibility of providing different fault-tolerance services during the same time quantum per period, as well as the same fault-tolerance service during more than one time quantum per period.

Other modelling and analysis problems needs to be addressed. The most urgent is the need to model *interacting* tasks, i.e. tasks that share resources through mutex semaphores and tasks interacting through remote procedure calls (RPC). We also need to better address the fault-recovery phase. We plan to combine our methodology with existing software techniques for fault-recovery (as checkpointing and primary-backup). Moreover, we are investigating on-line reconfiguration algorithms to recover as many tasks as it is possible after a fault.

References

[1] Hp nonstop advanced architecture - white paper. available at `http://h71028.www7.hp.com/ERC/downloads/NSAABusinessWP.pdf`.

[2] Powerpc750 lockstep facility - application note. available at `http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/292763D22B80DFEF872570C1006DF928/$file/750GX_Lockstep11-17-05.pdf`.

[3] Ppc405 lockstep system on ml310 - application note. available at `http://www.xilinx.com/bvdocs/appnotes/xapp564.pdf`.

[4] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM International Conference on Embedded software*, pages 95–103, Pisa, Italy, 2004.

[5] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *Proceedings of the 2003 international conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 170–177, San José (CA), U.S.A., 2003.

[6] S. K. Baruah. Partitioning real-time tasks among heterogeneous multiprocessors. In *Proceedings of the 33rd Annual International Conference on Parallel Processing*, pages 467–474, Montreal, Canada, Aug. 2004.

[7] R. C. Baumann. Soft errors in advanced semiconductor devices - part 1: The three radiation sources. *IEEE Transaction on Device and Materials Reliability*, 1(1):17–22, Mar. 2001.

[8] P. A. Bernstein. Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing. *Computers*, 21:37–45, Feb. 1988.

[9] E. Bini. *The Design Domain of Real-Time System*. PhD thesis, Scuola Superiore Sant'Anna, Pisa, Italy, Oct. 2004. available at `http://feanor.sssup.it/~bini/thesis/`.

[10] E. Bini and G. C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, Nov. 2004.

[11] M. Caccamo and G. Buttazzo. Optimal scheduling for fault-tolerant and firm real-time systems. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, pages 223–231, Hiroshima, Japan, Oct. 1998.

[12] X. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 26–35, Austin (TX) U.S.A., Dec. 2002.

[13] J.-Y. Le Boudec and P. Thiran. *Network Calculus*, volume 2050 of *Lecture Notes in Computer Science*. Springer, 2001.

[14] G. M. d. A. Lima. *Fault Tolerance in Fixed-Priority Hard Real-Time Distributed Systems*. PhD thesis, University of York, York, England, May 2003.

[15] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2):257–269, 2004.

[16] J. M. López, M. García, J. L. Díaz, and D. F. García. Utilization bounds for multiprocessor rate-monotonic scheduling. *Real-Time Systems*, 24(1):5–28, Jan. 2003.

[17] D. Mossé, R. Melhem, and S. Ghosh. Analysis of a fault-tolerant multiprocessor scheduling algorithm. In *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, pages 16–25, Austin (TX), U.S.A., Apr. 1994.

[18] S. Punnekkat. *Schedulability Analysis for Fault Tolerant Real-Time Systems*. PhD thesis, University of York, York, England, June 1997.

[19] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th Real-Time Systems Symposium*, pages 2–13, Cancun, Mexico, Dec. 2003.

[20] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398, Bethesda (MD), U.S.A., 2002.

[21] A. Srinivasan and J. H. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198, Montreal, Canada, 2002.