Fault-Tolerant Real-Time Scheduling using Chip Multiprocessor

Risat Mahmud Pathan
Department of Computer Science and Engineering
Chalmers University of Technology, SE-412 96, Göteborg, Sweden
risat@chalmers.se

Abstract

Failure to meet task deadline in safety critical real-time systems can be catastrophic. Moreover, fault tolerance is a crucial aspect of such systems if faults are likely. In this work, fault tolerance in real-time systems is proposed using time redundancy to mask at most F transient faults. Schedulability of a set of n preemptive real-time periodic tasks using Rate-Monotonic(RM) schedule in chip multiprocessor (CMP) is considered. Chip multiprocessor rather than uniprocessor is proposed to make more CPU time available before deadline of tasks. This paper addresses the issue of finding maximum number of tasks that can run in parallel at a particular time and also finds minimum number of processing cores required in a CMP, denoted by MinC, to make the real-time system task set schedulable. A real-time fault-tolerant algorithm FT-RT-CMP for scheduling tasks using MinC cores is also developed considering the worst case distribution of F transient faults.

1. Introduction

System with strict timing requirements are used in several applications like fly-by-wire, brake-by-wire, autopilot system and space shuttles, industrial process control and robots [1-3]. Some of these systems are considered as *hard real-time systems* where missing the deadline of a task can pose threat to human lives or environment. Moreover, when faults occur, recovery from the fault must be considered in such system. *Time redundancy* rather than *space redundancy* can be used due to cost, volume and space considerations. Time redundancy technique to mask faults at node level requires re-execution of tasks, known as Temporal Error Masking (TEM) [3-5]. Enough CPU time may not be available in the real-time schedule for such re-execution if the number of faults and task execution time is large. To make more CPU time available, an approach is made in this paper to achieve better schedulability using Rate-Monotonic(RM) scheduling for a set of n preemptive periodic tasks to tolerate a maximum of F transient faults using chip multi processor(CMP). In processor industry, the trend is now to build single chip multiprocessor(CMP) seeing the diminishing return from uniprocessor with higher transistor count [6-8]. In this paper, using such processor chip, with multiple cores having more computational power, scheduling of safety critical real-time application tasks to mask the effect of faults using time redundancy is proposed.

This paper in organized as follows: Section 2 provides the related work, section 3 presents the task model, and the inherent parallelism with such task model is discussed in Section 4. Task instances that can run in parallel at different cores of a CMP are defined formally in section 5. Section 6 presents algorithm *FT-RT-CMP* for scheduling tasks using CMP. Section 7 concludes this paper with a pointer to future work.

2. Related Work

In this work, I address the issue of tolerating transient faults that are temporary malfunctioning of the computing units. Such temporary malfunction can lead to an error in the system. The main source of transient faults is environmental disturbances like power fluctuations, electromagnetic interference and ionizing radiation by alpha particles. Several studies have shown that transient faults occur at much a higher rate than permanent faults [9-11]. In [9], measurements showed that transient faults are 30 times more frequent than permanent faults, while the work in [10] revealed that 83% of all faults were determined to be transient or intermittent. In some real-time systems such as satellites and space shuttles, transient faults occur at a much higher rate than in general purpose systems [11]. Such high occurrences of transient faults motivated me to the development of an approach to tolerate transient faults using time redundancy. Many approaches have been taken to address the use of time redundancy to tolerate transient faults [3-5, 12-14]. The work in [3] evaluates a real-time kernel that employs TEM for brake-by-wire applications where correct results increased from 81% to 89%. In [5], Ramos-Thuel presented an algorithm for fault-recovery based on concept of slack stealing. The work in [12] presented a recovery scheme using re-execution in the event of single or multiple errors. In [13], a temporal-redundancy-based recovery technique that tolerates transient task failures where tasks have different constraints is presented. An appropriate schedulability analysis for fault tolerant systems is made where recoveries of tasks may be executed at higher priority levels [14]. In my work here the execution of recovery tasks due to errors run as the same priority of the task in which the error is detected.

Since time redundant execution requires much CPU time, the use of multiple cores single chip multi processor(CMP) is proposed in this work. Building more powerful uniprocessors with increasing transistor counts has ceased due to limited instruction level parallelism, increased wire delay and latency to main memory access. Now trend is to accommodate many cores in the same die area, called Chip multi processor [6-8, 15]. On applications with large grained thread-level parallelism the multiprocessor microarchitecture performs 50–100% better than the wide superscalar microarchitecture [6]. Niagara chip multiprocessor increases application performance by improving throughput [7]. To application software, a Niagara processor will appear as 32 discrete processors with the OS layer abstracting away the hardware sharing [7]. If such processor is used for real-time scheduling, a total of 32 tasks can be scheduled in parallel using the support from operating system.

The low use of CMP today is because converting today's uniprocessor programs into multiprocessor ones is difficult. But in Section 4 of this paper we will see inherent parallelism in the real-time periodic task is the best target for CMP. For such inherent parallelism, the CMP is much more promising because it is already partitioned into individual processing cores [8]. To harness the benefit of CMP, applications must expose their thread-level parallelism to the hardware. This can be done by decomposing a program into parallel "tasks" and allow an underlying software layer to schedule these tasks on different threads [15]. Inspired by this approach, in this work task scheduling in CMP is addressed considering the potential parallelism within real-time periodic task set. In Section 6, the number of minimum cores in a CMP, denoted by *MinC,* required to make a task set schedulable is determined. In the next section, the task model used in this wok is presented.

3. Task model

The task set consists of *n* tasks, $\Gamma = \{\tau_1, \tau_{2,\dots}\ \tau_n\}$. Each task τ_i has a period T_i, and a relative deadline D_i is equal to task period T_i, worst case execution time C_i and priority P_i. The highest priority task has the lowest period. The length of the Planning Cycle (PC) in which the tasks repeat themselves iteratively is the least common multiple of all task periods.

$$PC = \text{L.C.M.}\{T_1, T_{2,\dots,}\ T_n\}.$$

Within one *PC*, one or more instances of task τ_i will execute. Each *task instance* is denoted by τ_{ij} where *j* is the j^{th} instance of task τ_i. The set of tasks that get ready at time t is denoted by *RD*(Γ,t) defined as:
$RD(\Gamma,t)=\{\tau_{ij} \mid \tau_i \in \Gamma \text{ and } t=(j-1)T_i \text{ for } j=1,2\dots (PC/T_i)\}$.
Γ_{all} is defined as the set of all task instances within PC. That is, $\Gamma_{all}=\{\tau_{ij} \mid i=1,2,\dots n \text{ and } j=1,2,\dots.(PC/T_i)\}$. All time units used in this work is integer values.

In this work, temporal error masking (TEM) for fault tolerance is used as follows: when a task is released, two *primary copies* of the task instance are run first. If an error is detected by comparison, or by error detection mechanism, *F* more *extra/recovery copies* of the same task instance are run and a majority voting is made to mask at most *F* errors. Figure 1 demonstrate this for *F*=2 and for a single task τ_1 with period T_1=10 and C_1=2.

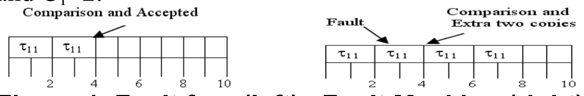


Figure 1. Fault free (left) , Fault Masking (right)

This paper is based on the assumption that, transient faults in different copies of the same task produce different outputs. As a result, the probability of having the same error in two primary copies is very small and error detection by comparison is possible. In next section, the inherent parallelism in tasks and how CMP could exploit such parallelism is discussed.

4. Task Level Parallelism

In Rate-Monotonic scheduling [16], the critical instance of the task sets is when all *n* tasks are released at time *t*=0. At time *t* task τ_i is released if $t=mT_i$ for some nonnegative integer m. When TEM is used, each of such primary copies of a task instance run twice in uniprocessor and can run in parallel in two cores of a CMP. Moreover, *F* more extra/recovery copies of a task instance are ready to run when error is detected. Each of the *F* extra/recovery copies can run in parallel in CMP if enough cores are available. These two scenarios show the inherent parallelism in application tasks of real-time system. In next subsection 4.1, how CMP can exploit such parallelism is demonstrated using an example.

4.1. Parallelism Exploitation by CMP

The inherent parallelism that can be exploited by CMP is demonstrated here using an example. Consider a real-time system with F=1 and two tasks τ_1 and τ_2 as in Figure 2 (left one). Also consider that, one of the two primary copies of τ_1 is in error. The Rate-Monotonic schedule is in Figure 2 (right one) with recovery copy running at t=2 to t=3.

τ_i	T_i	C_i
τ_1	3	1
τ_2	6	1

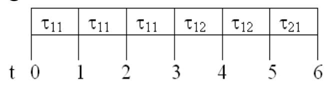
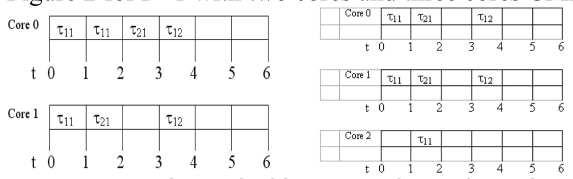


Figure 2. Task set (left), RM schedule (right)

The second instance of the first task τ_{12} (error free instance) finishes at t=5. The first instance of task τ_{21} does not have two time units with PC for execution of its two primary copies. So, the task set is not schedulable.

Considering the task level parallelism, the task model used in this work is an excellent target for CMP. Figure 3 shows two RM schedules for task set in Figure 2 for F=1 with two cores and three cores CMPs.



Core 0
τ_{11} τ_{11} τ_{21} τ_{12}
t 0 1 2 3 4 5 6
Core 1
τ_{11} τ_{21} τ_{12}
t 0 1 2 3 4 5 6
Core 0
τ_{11} τ_{21} τ_{12}
t 0 1 2 3 4 5 6
Core 1
τ_{11} τ_{21} τ_{12}
t 0 1 2 3 4 5 6
Core 2
τ_{11}
t 0 1 2 3 4 5 6

Figure 3: 2-Core (left) and 3-Core (right) rate monotonic schedule

In these CMPs schedules, the task τ_{21} is schedulable and other tasks have low response time and high slack is available that can be used to tolerate more faults or can be used to run other hard or soft aperiodic tasks.

Next question arises: how to generate such a schedule using CMP? How many tasks can run in parallel at any time instance t? In the next section, task instances that can run in parallel at time t are determined formally.

5. Parallel Task Instances

In the section 5.1 the set of primary copies of task instances, denoted by $PEX(t)$, that are ready to execute at time t in the fault free execution scenario is defined. In section 5.2 the set of extra/recovery copies of task instances, denoted by $FEX^F(t)$, that are ready to run at time t due to F faults is determined. In section 5.3 the set EX is defined combining the sets $PEX(t)$ and $FEX^F(t)$ to find the set of primary and extra task instances that are ready at to run within PC due to F faults. The set EX is used to find the minimum number of cores, denoted by $MinC$, to schedule all tasks using CMP with the scheduling algorithm $FT\text{-}RT\text{-}CMP$ is defined in section 6.

5.1 Primary Copies of Tasks at time t: PEX(t)

The set $PEX(t)$ contains triplets (a,τ_{ij}, b) such that the at time a task instance τ_{ij} need b unit of execution time in the uniprocessor RM schedule in the fault free case (where only two primary copies of τ_{ij} run). That is, $PEX(t) \subseteq \{t\}\times\Gamma_{all}\times N$. For example, $PEX(1)$ $=\{(0,\tau_{11},1), (0,\tau_{21},2)\}$ for task set in Figure 2.

Let the function $HP(PEX(t))=\tau_{lk}$ such that τ_{lk} is the highest priority task in set $PEX(t)$. For example, $HP(PEX(1))=\tau_{11}$ for the example task set in Figure 2. Lets formally define the function $PEX(t)$ as follows:

$$PEX(t)=\begin{cases} \{(t,\tau_{ij},2C_i) \mid \tau_{ij}\in RD(\Gamma,t)\} & \text{if } t=0 \text{ or} \\ & \text{if } t>0 \text{ and } t\neq PC \text{ and } PEX(t-1)=\emptyset \\ (PEX(t-1) - \{(a,\tau_{lk}, x)\}) \cup \{(a,\tau_{lk}, x-1)\} & \text{if } t=PC \text{ and } \tau_{lk}=HP(PEX(t-1)) \text{ and } x>1 \\ (PEX(t-1) - \{(a,\tau_{lk}, x)\}) & \text{if } t=PC \text{ and } \tau_{lk}=HP(PEX(t-1)) \text{ and } x=1 \\ (PEX(t-1) - \{(a,\tau_{lk}, x)\})\cup\{(a,\tau_{lk}, x-1)\} \cup \{(t,\tau_{ij},2C_i) \mid \tau_{ij} \in RD(\Gamma,t)\} & \text{if } t<PC \text{ and } \tau_{lk}=HP(PEX(t-1)) \text{ and } x>1 \\ (PEX(t-1) - \{(a,\tau_{lk}, x)\}) \cup \{(t,\tau_{ij},2C_i) \mid \tau_{ij} \in RD(\Gamma,t)\} & \text{if } t<PC \text{ and } \tau_{lk}=HP(PEX(t-1)) \text{ and } x=1 \end{cases}$$

Let consider the following task set in Table 1 where the PC=14. The PEX(t) is shown in Table 2.

Table 1: Task Set

τ_i	T_i	C_i
τ_1	7	2
τ_2	14	1

Table 2: PEX(t) for task in Table 1

t	PEX(t)
0	$\{(0,\tau_{11},4)(0,\tau_{21},2)\}$
1	$\{(0,\tau_{11},3)(0,\tau_{21},2)\}$
2	$\{(0,\tau_{11},2)(0,\tau_{21},2)\}$
3	$\{(0,\tau_{11},1)(0,\tau_{21},2)\}$
4	$\{(0,\tau_{21},2)\}$
5	$\{(0,\tau_{21},1)\}$
7	$\{(7,\tau_{12},4)\}$
8	$\{(7,\tau_{12},3)\}$
9	$\{(7,\tau_{12},2)\}$
10	$\{(7,\tau_{12},1)\}$
6,11,12,13,14	{}

From Table 2, it is clear that at time t=0, two primary copies of each task can run in parallel if four cores are available. Using set PEX, the execution finishing time of two primary copies of a task can be determined. For example, the finishing time of τ_{11} is 4 in Table 2 since this is the earliest t at which τ_{11} disappears from PEX(t). At t=14, new tasks are released but PEX(14)={} since only tasks released within PC are considered. The following functions are defined to deal with the faulty case in next section:

fin(τ_{ij}) = The execution finishing time of the primary two copies of task τ_{ij}. So, ***fin***(τ_{11})=4

pre((τ_{ij}))=τ_{lk} such that τ_{lk} finished immediately before τ_{ij} in fault free execution. So, ***pre***(τ_{12})= τ_{21}

slack(t_1,t_2)= k where k is the number of free slots between time t_1 and t_2 in a schedule . So, ***slack***(6,7)=1.

5.2 Extra/Recovery Tasks at time t: $FEX^F(t)$

In this section, the worst case fault distribution for a maximum of *F* faults as in [4] is considered. Similar to PEX(t), the set $FEX^F(t)$ contains triplets (p, q, r) where at time *p*, the extra/recovery copy of task instance *q* still needs a of total *r* units of execution time due to faults. In the following equations, the constant F is used for the number of maximum faults and the variable *f* is used to signify *f* number of faults, where $f \leq F$, to deal with scenario where less than F faults occurs. These extra copies, in $FEX^F(t)$ at time *t*, can be derived from the extra copies that are ready to run at the finishing time of the primary task copies. When a task τ_{ij} finishes execution of both primary copies at $t=fin(\tau_{ij})$, there are two cases to consider for worst case *f* faults distribution. *Case1* (defined by $Q^1_{ij}(f)$): all *f* faults caused *f* errors have already occurred before $fin(\tau_{ij})$. *Case2* (defined by $Q^2_{ij}(f)$: all (*f*-1) faults and consequent (f-1) errors have occurred before $fin(\tau_{ij})$ and a new fault is detected at $t=fin(\tau_{ij})$. The proof for the worst case scenarios based on case 1 and case 2 can be proved using induction on the number of faults. The total processing time required for all tasks in set $Q^1_{ij}(f)$ and $Q^2_{ij}(f)$ are defined using functions $W^1_{ij}(f)$ and $W^2_{ij}(f)$ respectively later. Before that, the set of triplets of extra tasks that are ready to run at $t=fin(\tau_{ij})$ due to f faults is defined by FEX^f_{ij} as follows:

$$FEX^f_{ij} = \begin{cases} \varnothing & \text{if } f=0 \\ \{(fin(\tau_{11}),\tau_{11}, F\times C_1)\} & \text{if } ij=11 \\ Q^1_{ij}(f) & \text{if } W^1_{ij}(f) > W^2_{ij}(f) \\ Q^2_{ij}(f) & \text{if } W^2_{ij}(f) > W^1_{ij}(f) \\ Q^K_{ij}(f) & \text{if } W^1_{ij}(f)=W^2_{ij}(f) \text{ and } Q^K \text{ has higher priority task than in } Q^{K(\text{mod } 2)+1} \end{cases} \quad \ldots \quad \ldots \quad \ldots(I)$$

The sets $Q^1_{ij}(f)$ and $Q^2_{ij}(f)$, and the functions $W^1_{ij}(f)$ and $W^2_{ij}(f)$ for *case1* and *case 2* are defined as follows:

Case 1: This case deals with scenario where all *f* errors have already occurred in tasks that has finished execution before $t=fin(\tau_{ij})$. The set $Q^1_{ij}(f)$ contains triplets (t, b ,c) such that at time $t=fin(\tau_{ij})$, the extra copy of task *b* with execution time *c* are ready to execute. $Q^1_{ij}(f)$ is defined as follows: $Q^1_{ij}(f) = SQ^s_{ij}(f)$ where $s=slack(fin(pre(\tau_{ij})), fin(\tau_{ij}))$ … … …(II)

$SQ^s_{ij}(f)$ is defined as follows:

$$SQ^s_{ij}(f)= \begin{cases} FEX^f_{pre(\tau ij)} & \text{if } s=0 \\ SQ^{(s-1)}_{ij}(f) & \text{if } SQ^{(s-1)}_{ij}(f) =\varnothing \\ (SQ^{(s-1)}_{ij}(f) - \{(a,\tau_{lk}, x)\}) \cup \{(a,\tau_{lk}, x-1)\} & \text{if } \tau_{lk}=HP(SQ^{(s-1)}_{ij}(f)) \text{ and } x>1 \\ SQ^{(s-1)}_{ij}(f) - \{(a,\tau_{lk}, x)\} & \text{if } \tau_{lk}=HP(SQ^{(s-1)}_{ij}(f)) \text{ and } x=1 \end{cases} \quad \ldots \quad \ldots \quad \ldots(III)$$

The function $SQ^s_{ij}(f)$ selects the highest priority extra copy of tasks in $FEX^f_{pre(\tau_{ij})}$ and reduces the execution time based on the slack available between the finishing time of the current and the previously completed task.

Case 2: All the (f-1) faults have occurred before $t=fin(\tau_{ij})$ and a new fault has occurred in τ_{ij} at $t=fin(\tau_{ij})$and the triplets in $Q^2_{ij}(f)$ is defined as follows:
$Q^2_{ij}(f)= Q^1_{ij}(f-1) \cup \{(fin(\tau_{ij}),\tau_{ij}, F \times C_i)\}$ … …(IV)

The amount of extra workload at time t is the sum of all execution times of the triplets in $Q^1_{ij}(f)$ and $Q^2_{ij}(f)$ and is defined by $W^1_{ij}(f)$ and $W^2_{ij}(f)$:
$W^1_{ij}(f)= \Sigma X$ such that $(a,\tau_{lk}, X)\in Q^1_{ij}(f)$ … …(V)
$W^2_{ij}(f)= \Sigma X$ such that $(a,\tau_{lk}, X)\in Q^2_{ij}(f)$ … …(VI)

Now, the set of triplets in $FEX^F(t)$, as explained at the beginning of this section, is defined as follows:
$FEX^F(t)=$

$$\begin{cases} \varnothing & \text{if } t=0 \\ FEX^F_{ij} & \text{if } t=\text{fin}(\tau_{ij}) \\ FEX^S_{ij}(t) & \text{if } fin(\tau_{ij}) < t < fin(\tau_{lk}) \text{ and } pre(\tau_{lk})= \tau_{ij} \text{ and } s=slack(fin(\tau_{ij}),t) \\ (FEX^F(t-1) - \{(a,\tau_{lk}, x)\}) \cup \{(a,\tau_{lk}, x-1)\} & \text{if } \tau_{lk} =HP(FEX^F(t-1)) \text{ and } x>1 \text{ and } FEX^f(t-1)\neq\varnothing \text{ and } [t-1,t] \text{ is a free slot} \\ FEX^F(t-1) - \{(a,\tau_{lk}, x)\} & \text{if } \tau_{lk} =HP(FEX^F(t-1)) \text{ and } x=1 \text{ and } FEX^F(t-1)\neq\varnothing \text{ and } [t-1,t] \text{ is a free slot} \\ FEX^F(t-1) & \text{Otherwise} \end{cases} \quad \ldots \quad \ldots \quad \ldots(VII)$$

The function $FEX^S_{ij}(t)$ selects the highest priority extra copy of tasks in FEX^F_{ij} and reduces the execution time based on the slack available between the finishing time of the current task and the previously completed task. $FEX^S_{ij}(t)$ is defined as follows:

$$FEX^S_{ij}(t)= \begin{cases} FEX^F_{ij} & \text{if } s=0 \\ \varnothing & \text{if } FEX^{(s-1)}_{ij}(t)= \varnothing \\ (FEX^{(s-1)}_{ij}(t)-\{(a,\tau_{lk},x)\})\cup\{(a,\tau_{lk},x-1)\} & \text{if } \tau_{lk}=HP(FEX_{ij}^{s-1}(t)) \text{ and } x>1 \\ (FEX^{(s-1)}_{ij}(t)-\{(a,\tau_{lk},x)\} & \text{if } \tau_{lk}=HP(FEX_{ij}^{s-1}(t)) \text{ and } x=1 \end{cases} \quad \ldots \quad \ldots \quad \ldots(VIII)$$

Now, for *F*=2 the extra/recovery task instances that are ready to execute at time t for the example task set in Table 1 is determined using equations I-VIII. For *t*=0 to *t*=4 fault occurrence is not detected since two primary copies has not finished execution. So, using the first and last conditions of equation (VII), $FEX^2(0)= FEX^2(1)= FEX^2(2) =FEX^2(3)=\varnothing$. At t=4, using second condition of (VII), $FEX^2(4)=FEX^2_{11}$ since $fin(\tau_{11})=4$. Using the second condition of equation (I), $FEX^2_{11}=\{(fin(\tau_{11}),\tau_{11},F\times C_1)\}=\{(4,\tau_{11},4)\}$. At t=5, using the third condition of (VII), $FEX^2(5)= FEX^S_{11}(5)$ as $fin(\tau_{11})=4 < t < fin(\tau_{21})=6$ and $pre(\tau_{21}) =\tau_{11}$ and $s=slack(fin(\tau_{ij}),t)=0$. Now using the first condition of (VIII), $FEX^0_{11}(5)=FEX^2_{11}$. FEX^2_{11} is known at t=4 and we have $FEX^0_{11}(5)= \{(4, \tau_{11}, 4)\}$. At t=6, using second condition of VII we have $FEX^2(6)=FEX^2_{21}$ since $fin(\tau_{21})=6$. Following the equation (I), the triplets in FEX^2_{21} is either $Q^1_{21}(2)$ or $Q^2_{21}(2)$. Case 1: All 2 faults have occurred before. So, $Q^1_{21}(2)= SQ^s_{21}(2)= SQ^0_{21}(2)$ where s = slack($fin(pre$(ij)),fin(ij)) =slack(4,6)= 0 using (II). $SQ^0_{21}(2)= FEX^f_{pre(\tau ij)}$ since s=0 using the first condition of (III). So, $SQ^0_{21}(2)= FEX^f_{pre(\tau ij)}=FEX^2_{11}=$ $\{(4, \tau_{11}, 4)\}$ and we have, $Q^1_{21}(2)=SQ^0_{21}(2)= \{(4, \tau_{11}, 4)\}$. The total workload at t=6 is $W^1_{21}(2)=4$ using (V). Case 2: One fault occurred before and a new fault has occurred in τ_{21}. Using (IV), $Q^2_{21}(2)= Q^1_{21}(1) \cup \{(fin(\tau_{21}),\tau_{21},2 \times 1)\} = Q^1_{21}(1) \cup \{(6,\tau_{21},2)\}$. Using equation (II) and (III), we have $Q^1_{21}(1)=SQ^0_{21}(1)$since s =slack(fin(pre(τ_{21})),fin(τ_{21})) =slack(fin(τ_{11}),fin(τ_{21}))= slack(4,6) =0. $SQ^0_{21}(1)= FEX^1_{11} =\{(fin(\tau_{11}),\tau_{11}, F\times C_1)\}$ $=\{(4, \tau_{11}, 4)\}$. So, we have, $Q^2_{21}(2)= Q^1_{21}(1) \cup \{(fin(\tau_{21}),\tau_{21}, 2 \times 1)\}= Q^1_{21}(1) \cup \{(6,\tau_{21},2)\}= SQ^0_{21}(1) \cup \{(6,\tau_{21},2)\} =FEX^1_{11} \cup \{(6,\tau_{21},2)\}=\{(4, \tau_{11}, 4), (6,\tau_{21},2)\}$. And the total work load at t=6 is $W^2_{21}(2)=6$. Since $W_{21}{}^2>W_{21}{}^1$ at t=6, $FEX^2_{21}= Q^2_{21}(2) =\{(4, \tau_{11}, 4), (6,\tau_{21},2)\}$ using the fourth condition of (I). At t=6, $FEX^2(6)=\{(4, \tau_{11}, 4), (6,\tau_{21},2)\}$. Other $FEX^2(t)$ are as follows (can be found using equations I-VIII):

$FEX^2(7)=\{(4,\tau_{11},3),(6,\tau_{21},2)\}$ $FEX^2(8)=\{(4, \tau_{11}, 3), (6,\tau_{21},2)\}$
$FEX^2(9)=\{(4,\tau_{11},3),(6,\tau_{21},2)\}$ $FEX^2(10)=\{(4,\tau_{11}, 3), (6,\tau_{21},2)\}$
$FEX^2(11)=\{(4,\tau_{11},3),(11,\tau_{12},4)\}$
$FEX^2(12)=\{(4,\tau_{11},2),(11,\tau_{12},4)\}$
$FEX^2(13)= \{(4,\tau_{11},1), (11,\tau_{12},4)\}$ $FEX^2(14)= \{(11,\tau_{12},4)\}$

In the next subsection 5.3 the set *EX* is defined that combines tasks from *PEX* and *FEX* to find the triplets when tasks are just released and ready to execute.

5.3 Set *EX*: Combining PEX(t) and FEX^F(t)

The set EX contains the triplets when a task instance is just released as primary copy or recovery copy in case of worst case distribution of *F* faults. Tasks of the triplets in EX have only $2\times C_i$ or $F\times C_i$ execution time whereas triples in PEX or FEX may have values less than $2\times C_i$ or $F\times C_i$ for primary and extra copies correspondingly. EX is defined as follows:

$EX=\{(t,\tau_{ij},X) | (X=2\times C_i$ and $(a, \tau_{ij} ,X)\in PEX(t))$ or
$(X=F\times Ci$ and $(a, \tau_{ij} ,X) \in FEX^F(t))$ for $t=0,1\ldots PC\}$

For the task set in Table 1, the set EX for *F*=2 is as follows: $EX=\{(0,\tau_{11}, 4), (0,\tau_{21}, 2), (4,\tau_{11}, 4), (6,\tau_{21}, 2), (7,\tau_{11}, 4), (11,\tau_{11}, 4)$. It is not the case that all task instances have to run additional *F* copies. If the *F*=1, for the task in Table 1, $EX=\{(0,\tau_{11}, 4), (0,\tau_{21}, 2), (4,\tau_{11}, 2), (7,\tau_{12}, 4), (11,\tau_{12}, 4)\}$ where no extra copy for τ_{21} needs to run. In next section, the *FT-RT-CMP* algorithm is developed using set *EX* to find the minimum number of cores, denoted by *MinC,* to successfully schedule the task set *EX.*

6. Scheduling Algorithm in CMP:

The *FT-RT-CMP* algorithm, using the ***while*** loop in line 3-44 schedules tasks in set *EX* using *NP* number of cores (line 2). If a task is not schedulable using *NP* cores, *NP* is increased by 1 (line 4). The *MinC* is set at line 7 each time the while loop (line 3-44) starts. For each of the *NP* cores total *PC* free time slots are available and simulated using the 2D array *Slot* at line 10 (initially set to "free"). The ***while*** loop at line 12-41, schedules all the tasks in *EX* using total *NP* cores.

Algorithm: FT-RT-CMP(*Set of Triplets* EX)

```
1 MAX_NP=Maximum number of cores available in a CMP
2 NP=0
3 While (NP < MAX_NP)
4  Label 1: NP=NP+1
5          Set of Triplets EX_Temp=EX
6          //Minimum Number of cores to start the schedule
7          int MinC=NP
8          // For Each of NP cores total PC time slots are
9          //available and set to "free" slot
10          2D-Array of type Task Slot[NP][PC]= {"Free"}
11         //This loop schedules all individual task $\tau_{ij}$
12         While (Ex_Temp $\neq \varnothing$)
13            Find the highest priority task $\tau_{lk}$ in EX_Temp
14            For each (a, $\tau_{lk}$ ,b)$\in$ EX_Temp
15                 Find the number of all copies in TotalCopy
16            Find ReleaseTime and DeadLine for task $\tau_{lk}$
17            //All copies of the highest priority task is
18               //scheduled in the following loop
19              Label 2: While (TotalCopy$\neq$0)
20                        SeqTimeUnit=$C_l$
21                 Label 3: For i=ReleaseTime to Deadline
22                          For P=1 to NP
23                            If (Slot[P][i]="Free") then
24                               Slot[P][i]= $\tau_{lk}$
25                               SeqTimeUnit= SeqTimeUnit -1
26                               If SeqTimeUnit=0 then
27                                 TotalCopy=TotalCopy-1
28                                 Goto Label 2
29                               Else
30                                  Goto Label 3 for next i
31                               End if
32                            End if
33                          End For
34                        End For
35                 If SeqTimeUnit$\neq$0 then
36                    Print "$\tau_{lk}$ is not schedulable in NP cores"
37                   If NP=MAX_NP then
38                     Print "Task set not schedulable" and Exit
39                       Else Goto Label 1 End if
40              End While
41         End while
42         Print "The task set is Scheduleable"
43         Return Slot and MinC
44  End While
```

Figure 4. Fault tolerant Schedule using CMP

In line 13-15, the highest priority task τ_{lk} is extracted from set *EX* and total number of this task copies that can run in parallel is calculated in variable *TotalCopy*. The release and deadline of τ_{lk} is determined during which task τ_{lk} is scheduled in the ***while*** loop at line 19-40. Note that, each task copy of τ_l should have sequential C_l time unit in the schedule, is stored in *SeqTimeUnit* variable at line 20. The two nested ***For*** loops at line 21-34 check for C_l units of sequential free slots in array *Slot* to schedule the task in single or multiple cores. If *SeqTimeUnit* becomes 0, the task copy is scheduled successfully, *TotalCopy* is decreased by 1 and the ***while*** loop at line 19 iterates for next copy of the same task if available, otherwise, the ***while*** loop at line 12 starts with the next priority task in set *EX*. If *SeqTimeUnit* is not 0 when the nested loop in lines 21-34 is over, the task τ_{lk} can not be scheduled using *NP* number of cores and *NP* is increased by one (line 46 and line 4). If the set *EX* is empty, the task set is schedulable using *MinC* cores and the array *Slot* representing the schedule is returned from line 43. Figure 5 is the real-time fault-tolerant schedule for task in Table 1 for *F*=2. *MinC=2* is determined by the FT-RT-CMP algorithm.

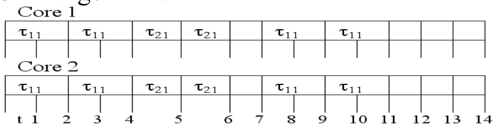


Figure 5. The FT-RT-CMP Schedule for task set in Table 1 using two cores (*MinC*=2)

7. Conclusion

The inherent parallelism within real-time periodic tasks are exploited by CMP in this paper by scheduling the tasks in set *EX* by using the *FT-RT-CMP* algorithm that finds the minimum number of cores *MinC* required for a feasible rate-monotonic fault-tolerant schedule while masking a maximum *F* transient faults. Time redundant execution for fault masking is addressed using CMP's ability to make more CPU time available. More slack become available in the schedule and real-time task set not schedulable in uniprocessor becomes schedulable using CMP even if *F* faults occur. For some task set with high task level parallelism the maximum number of cores can be provided to make the task set schedulable. But providing more cores requires more money and may not be available from CMP industry. In the future, chip microprocessors are expected to support beyond 100 simultaneous threads and can run 100 real-time system tasks in parallel. In future, exact schedulability conditions and other scheduling algorithm like EDF can be considered for chip multiprocessor.

REFERENCES

[1] D. Briere and P. Traverse, "AIRBUS A320/A330/A340 electrical flight controls- A family of fault-tolerant systems", FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing, 22-24 June 1993, Toulouse, France, 1993.

[2] L. Andrade and C. Tenning, "Design of Boeing 777 electric system", IEEE Aerospace and Electronics Systems Magazine, vol. 7, pp 4-11, 1992.

[3] Joakim Aidemark, "Node-Level Fault Tolerance for Embedded Real-Time Systems" Ph. D. Thesis, Chalmers University of Technology, 2004.

[4] Frank Liberato, Rami Melhem, and Daniel Mosse. "Tolerance to multiple faults for aperiodic tasks in hard real time system", IEEE Trans. Computers 49(9):906-914, 2000.

[5] S. Ramos-Thuel. "Enhancing Fault tolerance of Real-time systems through Time redundancy", Ph. D. Thesis, Carnegie Mellon University, May 1993.

[6] L. Hammond, B. Hubbert, M. Siu, M. Chen, K. Olukotum, "The Stanford Hydra CMP", IEEE MICRO Magazine, March-April 2000, and presented at Hot Chips 11, August 1999.

[7] P. Kongetira, K. Aingaran, K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor" , IEEE Micro, Vol 25, No 2, Mar.Apr. 2005, pp 21-19

[8] L. Hammond, B. A. Navfeh, K. Olukotun, "A Single-Chip Multiprocessor", IEEE Computer Special Issue on "Billion-Transistor Processors", September 1997.

[9] D. P. Siewiorek, V. Kini, H. Mashburn, S. McConnel, and M. Tsao, "A Case Study of C.mmp, Cm*, and C.vmp: Part 1: Experiences with Fault Tolerance in Multiprocessor Systems", Proceedings of the IEEE, 66(10):1178-1199, Oct. 1978.

[10] R. K. Iyer, D. J. Rossetti and M. C. Hsueh, "Measurement and Modelling of Computer Reliability as Affected by System Activity", ACM Trans. On Computer Systems, 4(3): 214-237, Aug. 1986.

[11] A. Campbell, P. McDonald, and K. Ray. "Single Event Upset Rates in Space", IEEE Trans. On Nuclear Science, 39(6): 1828-1835, Dec, 1992.

[12] S. Ghosh, R. Mehlem, D. Mosse and J. S. Sarma. "Fault tolerant rate monotonic scheduling", Journal of Real-Time Systems, 15(2), September 1998.

[13] N. Kandasamy, J. P. Hayes, and B.T. Murray, "Tolerating Transient Faults in Statically Scheduled Safety Critical Embedded Systems", Proc. 18th TEEE symposium Reliable Distributed System(SRDS), pp. 212-221, 1999.

[14] A. Burns, R. Davis, and S. Punnekkat, "Feasibility Analysis of Fault Tolerant Real Time Task Sets", In 8th Euromicro Workshop on Real-Time Systems, Jun 1996.

[15] Kumar, S.; Hughes, C.J.; Nguyen, A.; Kumar, A. "Architectural Support for Fine-Grained Parallelism on Multi-core Architectures" Intel Technology Journal. Vol. 11 Issue 03 , August 2007.

[16] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment", Journal of ACM, 20(1): 40-61, 1973.