基于服务体/执行流模型的 MiniCore 系统的容错设计①

杨金彪[1], 陈香兰[2]

[1](中国科学技术大学 软件学院, 合肥 230027)
[2](中国科学技术大学 计算机科学与技术学院, 合肥 230027)

摘 要: 空间环境中的计算机系统要求高可靠性. 针对存储受限的空间嵌入式实时系统, 从内存可靠性和任务容错调度两个角度出发, 提出了一种两级容错设计方案. 该方案由系统级的周期性内存检错纠错机制和任务级的一种改进的主/副版本容错调度机制组成. 方案的实验验证在一款基于服务体/执行流模型(SEFM)设计的嵌入式操作系统 MiniCore 中进行. 加入两级容错机制后, 内存数据准确性得到保证, MiniCore 内核代码空间增加了约 33%, 时间性能指标略微下降, 任务的执行成功率和调度质量显著增加.

关键词: 空间嵌入式实时系统; 容错调度; 两级容错设计; 服务体/执行流模型

Fault-Tolerant Design of MiniCore Operating System Based on Servent/Exe-Flow Model

YANG Jin-Biao[1], CHEN Xiang-Lan[2]

[1](Department of Software Engineering, University of Science and Technology of China, Hefei 230027, China)
[2](Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)

Abstract: Computer systems require high reliability in space environment. Contrary to limited storage space for embedded real-time systems, considering the memory reliability and Fault-tolerant task, we propose a two level of fault-tolerant solution. The program consists of a periodic memory detecting and an error correcting mechanism of system-level and an improved master/slave version scheduling mechanisms of task-level. The experimental verification of the program is carried out in an embedded operating system named MiniCore which based on a service body/execution flow model (SEFM). The introduction of fault-tolerant mechanism increases the code size of MiniCore kernel by 33% and ensures the accuracy of the memory data, with the system's time performance declining slightly and the success rate and scheduling performance of task execution improving significantly.

Key words: space embedded real-time systems; fault-tolerant scheduling; two level of fault-tolerant design; service body/execution flow model

实时系统是计算机应用技术的一个重要方向. 广泛应用于军事、航空航天、工业控制等领域. 典型的应用包括星载小卫星操作系统、汽车控制系统、网络通信系统、电子产品等. 而且在一些硬实时系统[1]中, 如航空航天和工业过程控制. 多种任务定期调用和执行, 以便在确定的最后期限前完成一个共同的功能. 在这些硬实时系统中, 任何微小的错误都可能造成严重的损失.

在空间嵌入式实时系统上主要是一些周期性的任务, 实时性要求高, 任务必须在规定的时间范围内正确地响应外部物理过程的变化, 一旦任务执行时间超过时限, 就会导致任务执行失败, 引起难以预测的严重后果.

但是程序员不可能设计一种完全无错的系统, 为了提高系统的可靠性, 必须采用一些手段来解决系统错误. 这些手段可分为错误预防、错误验证、错误预

① 基金项目:国家"核心电子器件、高端通用芯片及基础软件产品"重大专项(2012ZX01034001-001);国家自然科学基金(61379040,61272131)
收稿时间:2015-04-20;收到修改稿时间:2015-09-09

报和容错四种[2]. 在这几种手段中, 使用容错来提高系统的可靠性的是安全有效的.

1 相关工作

容错为操作系统提供了一种安全可靠性保障机制, 其目的是保证在系统出现错误时也能够继续提供安全可靠的服务. 针对不同的容错技术能够处理不同类型的缺陷和错误.

1.1 操作系统出错情况分析

运行于空间环境中的计算机操作系统主要会遇到以下三类错误:

1) 环境上的影响, 如单粒子翻转错误. 单粒子事件是外太空中的高能量微小粒子造成的, 它们打击星载计算机硬件后会造成硬件位翻转, 从而使而主存区的数据变化, 并最终导致指令或数据出错, 影响系统中指令和软件的执行.

2) 硬件错误: 如部件故障等. 部件故障可能是由于生产的时候造成的, 也可能是由于在使用中不断地磨损所造成的.

3) 软件错误: 如软件设计错误、任务执行结果错误等. 软件缺陷包括需求错误、功能性错误、软件性能错误、软件系统结构错误等; 任务执行结果出错是因为在实时系统中, (Real-time operating system)任务未能在截止期前产生正确的结果.

1.2 容错技术研究

根据冗余资源形式的不同, 可以将冗余方法分为以下四类:

1) 硬件冗余: 指为操作系统中硬件模块进行备份, 如双机冗余、多模冗余等. 常用的硬件冗余方法是三模块表决系统.

2) 软件冗余: 指在系统中为单个功能设计多个不同的软件模块, 是的一小部分模块出错时, 其功能可以由其它的软件模块实现. 常用的有多版本编程技术、恢复快技术、检测点技术、前向恢复和后向恢复技术等.

3) 信息冗余: 为了解决信息在运算或传输过程中的错误, 为传输数据外加一部分冗余信息码, 用以编码、校验. 常用方法有奇偶编码、海明编码、剩余码编码技术、循环校验码等.

4) 时间冗余: 指系统预留部分空闲的时间, 在系统出现错误后, 可以有时间进行错误处理和恢复操作. 在这里, 一般是在预留的时间里再次运行出错任务或者其备份任务, 以期获得正确的或降级的运行结果.

1.3 容错实时调度算法研究

实时系统要求任务在正确执行或者执行出错后都能够在截止期之前得到正确结果. 普通的实时调度算法只考虑任务正常工作是的时间要求. 但是任务在执行的过程中可能出错或者超过时限, 保证实时系统中任务的正确执行, 必须有容错实时调度算法的支持, 否则, 任务的执行结果是不确定的, 在执行的过程中可能出现错误.

根据系统在调度时间的不同,分为离线的容错实时调度算法和在线的容错实时调度算法. 在离线容错实时调度中,任务的调度方案在操作系统运行前以及确定. 此类调度算法的优点是确定性好,运行时间开销小, 但是空间开销大, 灵活性不好,仅适用于简单的容错实时系统.

而在线容错实时调度则是根据任务的优先级来确定调度顺序的. 按实时任务的周期特性, 可以分为周期性和非周期性容错实时调度算法; 按实时任务是否可抢占, 分为可抢占和不可抢占的容错实时调度算法.

1.4 容错小结

容错实时系统[3]的研究主要集中在两个方面. 一是关于多种容错策略的研究, 如 N 版本冗余技术、多模冗余技术、恢复块技术[4-6]等等. 二是改进容错实时调度算法, 确保实时任务在正常运行和遇到错误时, 能够在时限到来之前获得正确的输出. 如罗威等人[7]提出了基于延迟主动副版本的容错调度算法.

但这些容错方法效果并不是很理想, N 版本冗余一般适用于多处理器, 并且系统空间资源开销大; 多模冗余硬件要求高; 另外, 仅使用单一的容错方法并不能让系统在时间和空间上作好平衡.

为此, 本文针对空间嵌入式实时系统 MiniCore 设计了一个两级容错设计方案, 在系统级提出基于内存保护的容错机制, 使用内存预分配策略和对内存数据进行周期性检错纠错, 保证系统内存数据的准确性; 在任务级提出基于任务调度的容错机制, 提高用户任务的执行成功率和调度质量.

2 基于SEFM的MiniCore操作系统介绍

2.1 服务体/执行流模型(SEFM)介绍

使用进程/线程模型设计的操作系统, 比如宏内核操作系统或者微内核的操作系统, 都或多或少存在一些缺点, 如进程之间通信需要跨越地址空间, 降低通

信的效率; 频繁的上下文切换与现场保存大大降低了系统的效率等等.

为了改进进程/线程模型的缺点, 李宏等人提出了服务体/执行流模型[8]. 服务体/执行流模型的结构示意图如图 1 所示. 服务体/执行流模型由服务体和执行流两类机制组成.

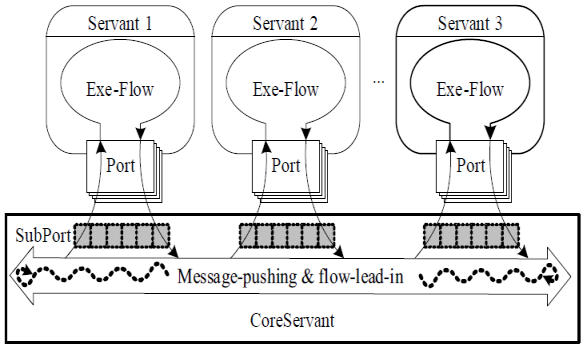

图 1 基于 SEFM 的操作系统的基本结构

服务体是系统的基本组成单位,包含了对数据的一些基本抽象以及未来能够完成某一功能的代码和数据. 用户程序和驱动程序都以服务体的形式存在. 其中核心服务体提供内存管理、服务体管理、消息通信、小端口调度、中断、时钟等基础服务. 在逻辑功能上相对于微内核结构中的内核.

执行流代表了CPU对机器码的执行, 是CPU沿着指令计数器指示的执行顺利执行指令而形成的连续轨迹. 系统中的每个 CPU 提供一条执行流. 执行流是连续的, 存在于系统运行的整个生命周期,不会被阻塞、挂起, 停止或者撤销. 可以说执行流是比线程更加基本的抽象.

服务体间使用基于消息的通讯机制, 在执行流推动下处理消息. 使得服务体/执行流模型具有即时性、高效性和安全性等优点[9].

2.2 MiniCore 体系结构

MiniCore 是基于 SEFM 实现的一款嵌入式操作系统原型, 根据功能划分为 3 个层次, 分别是基本内核层、应用服务层和运行环境层. 使用这种层次化的内核设计可以提高系统的可扩展性.

基本机制层提供系统运行的一些基础机制, 包括有关 SEFM 的抽象、内存管理、消息通信、服务体管理、中断、时钟等机制.

应用服务层提供一些系统服务功能, 常用的有 IO 管理服务、文件系统服务、对象管理服务、以及驱动服务等.

运行环境层提供程序运行环境和编程模型. 其中, Linux 运行环境等则用兼容现有的操作系统二进制应用程序; 在本地运行环境则提供服务体/执行流模型的编程环境.

MiniCore 作为一款空间嵌入式实时系统, 特点如下:

(1) 支持任务优先级继承, 解决优先级翻转

(2) 支持抢占式任务调度, 使用基于消息的通信机制

(3) 系统实时性好

(4) 内核可裁剪

(5) 可扩展性和可维护性高

3 MiniCore操作系统的容错设计

大型操作系统系统资源充足, 可以使用复杂的容错技术. 例如, IBM 的 MVS/ XRF 采用的是恢复模式, Tandem/GUARDIAN 采用主/从处理器模式等[10]. 但是, 空间嵌入式实时系统在空间和时间上的系统资源十分有限, 因此必须在系统资源的使用上和容错功能的实现上作出平衡.

空间系统的主要错误来自于与存储管理相关的操作, 大约占所有错误的 61%, 如所有针对单粒子翻转可能造成的错误. MiniCore 将数据的存储和数据的计算相互分离, 对内存数据的保护尤为重要. 同时, 系统任务不仅有时限要求, 在任务执行出错后还要有相应的容错机制. 在保证可靠性的基础上避免使用过于复杂的容错技术, 防止造成资源开销过大. 针对系统自身和任务的容错需求, 以及空间环境因素的考虑, 本文提出一种两级软件容错方案. 如图 2 所示.

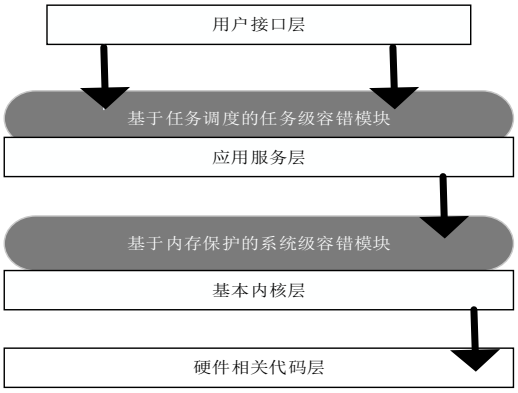

图 2 两级容错结构

在基本内核层实现系统级的容错策略, 增加了对内存数据的保护, 通过内存预分配周期性内存检错纠错机制, 保证内存数据的准确性; 在应用服务层实现任务级的容错策略, 提出了一种改进的容错任务调度算法, 用以提高任务的执行成功率和调度质量.

4 基于内存保护的系统级容错设计

4.1 内存预分配策略

在 MiniCore 系统的设计和实现当中, 采用了准静态内存分配策略,避免了在 RAM 中传递和使用内存指针. 因为在某一状态下, 内存的分配是可以确定的而无需动态修改的, 只有在多个系统状态之间切换时可能伴随着内存分配的切换. 另外, 在星载小卫星系统中, 主要处理一些预知的任务, 因此我们可以在系统的设计中, 预先对各个任务、服务和其它系统构件所需的内存进行划分和分配, 并将这种分配和划分具体实现在代码的实现中. MiniCore 内存分区设计如图 3 所示.

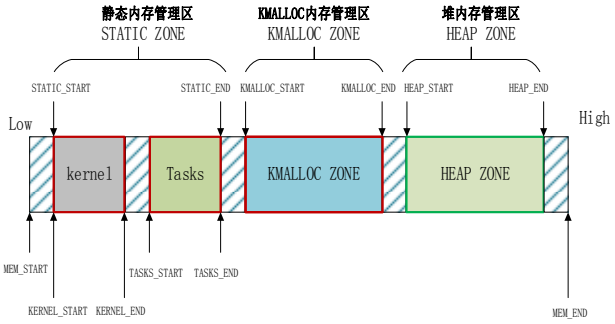

图 3　MiniCore 内存分区

通过静态联编得到的内核镜像加载到静态内存管理区(STATIC_ZONE), 包括内核的代码段、数据段等(数据段中包含一部分预先分配好的静态数据结构, 如空闲任务控制块、信号量、消息队列等). 而对于系统运行时对内存的动态请求则由 KMALLOC 内存管理区和堆内存管理区满足. 这样设计, 降低了恶劣环境对系统的可能影响.

4.2 基于信息冗余的周期性内存检错纠错机制

编码容错技术[11]非常适合于内存数据的容错. 该技术能够对内存数据进行检错和纠错, 能够在短时间内恢复数据, 具有很好的实时性, 非常有利于卫星数据传输.

海明码具有简单、高效的特性, 很多系统中都使用它进行校验纠错. 为了使编码校验的功能对整个系统尽可能的透明, 我们采用把原始数据和校验码分离的方法, 保证原始数据的完整连续性. 在本系统中, 本文使用汉明编码, 内存数据区中每 16 位数据字使用 6 位校验字与之对应, 并在内存管理区单独划分一个冗余代码管理区, 用以以存放校验信息, 来支持编码的生成或检查.

对于内存数据的编码校验, MiniCore 提供了两个基本的服务原语, Code(编码)服务和 Decode(解码)服务, Code 服务用以对内存数据进行计算并存储校验码, 而 Decode 服务则进行内存数据的检错纠错.

对于内存的关键数据, 在为其加入检错纠错机制后, 可以单独用一个周期性任务来定时的更新, 纠正内存关键数据中可能存在的错误. 定义此任务为 task_refresh,任务周期为 T, 该任务的主要主要执行以下 4 个操作:

a.数据准备: 从记录的内存地址区域中将所需数据读出.

b.编码校验: 进行校验, 判断编码的正确性.

c.纠错处理: 如果发现编码有误, 纠正错误的编码. 没有错误则返回

d.更新恢复: 更正内存区中的内容.

5 基于容错任务调度的任务级容错设计

5.1 容错调度模型:

主/副版本技术是 Liestman[12]等人首先提出的. 每个任务有两个独立的版本, 分为主版本(Primary)任务和副版本(Alternative)任务. 主版本: 包含更多的功能性(因此更复杂), 产生质量好的结果, 但是其可靠性不能保证. 副版本: 只有最小要求的功能(因此简单), 产生较少的精确性, 因此其可靠性得以保证.

本文考虑一个典型的由单处理机和周期性任务集构成的实时系统, 其形式化描述如下:

1) 一组实时周期任务

任务集 $\Gamma=\{\tau_1,\tau_2,\tau_3\ldots,\tau_n\}$,

每个任务 τ_i 有一个周期 T_i.

2) 每个任务 τ_i 有两个独立的版本, 主版本 P_i 和副版本 A_i.

3) P_i 有一个运行时间 p_i, A_i 有一个运行时间 a_i, 一般的, $p_i>a_i$ for $1<i<n$.

4) 计划周期 T 是 T_1, T_2, T_3, …, T_n 的最小公倍数,

那么 $n_i=T/T_i$ 就是任务 τ_i 在一个计划周期内执行的次数

主/副版本任务调度算法需要预先为副代版本分配时间, 当主版本失败或无法按时完成时就要运行相应的替代版本. 所以定义 notification_time, 表示副版本的最晚开始执行时间. 它是根据 backward-RM[13]算法, 由后往前来计算副版本的最晚开始调度时间的.

举例如下, 对于一个周期性任务集 $\Gamma=\{\tau_1,\tau_2\}$, task1(T_1,p_1,a_1=(5,2,1)),task2(T_2,p_2,a_2=(6,2,2)). 那么其计划周期 T=30(计划周期等于 T_1 和 T_2 的最小公倍数). task1 的 notification_time 是 4,9,14,19,24,29, 即到了这些时刻, 若主版本执行失败或未执行完, 此时必须执行副版本(否则, 副版本也将会没有足够的执行时间完成任务). task1 和 task2 的副版本调度时刻如图 4 所示.

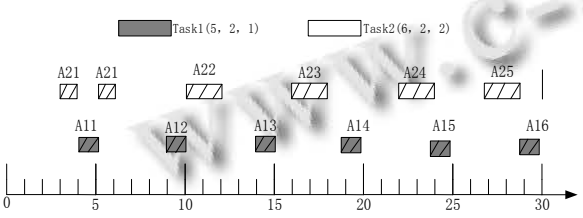

图 4 task1 和 task2 的副版本调度时刻

5.2 调度目标及分析

本文主要有两个目标: 一是保证在任务截止期之前每个任务的主版本或副版本被成功执行. 二是完成尽可能多的主版本任务, 获得更好的计算质量.

H. Chetto and M. Chetto[14]等人提出了一种单处理机上主/副版本技术的基本容错调度算法. 该算法能有效的保证任务(主任务或副版本任务)在截止期限内完成, 但是, 该算法在某些情况下会遇到严重的系统退化问题, 即一个主版本任务的失败可能会影响接下来的多个任务.

举例如下, 假设一个周期性任务集 $\Gamma=\{\tau_1,\tau_2\}$, task1(T_1,p_1,a_1=(9,5,2)),task2(T_2,p_2,a_2=(14,4,3)),P11 表示 task1 的主版本任务的第一次执行, P12 表示 task1 的主版本任务的第二次执行, P21 表示 task2 的主版本任务的第一次执行. 调度情况如图 5 所示.

从图 5 中我们可以看到, 假设 P11 在时刻 5 失败, 那么 A11 在时刻 7 到 9 执行, P12 在时刻 9 到 11 执行, 但是在时刻 11, A21 来临, 抢占 P12, P12 在时刻 14 到 16 执行, 在时刻 16, A12 来临, 抢占 P12, 如图所示, 至少三个接连的主版本任务被 P11 的失败所影响了, 被迫放弃执行. 这个例子展示了基本的主/副版本调度算法的劣势: 它缺少一些功能来保护接下来的作业不被先前的失败所影响.

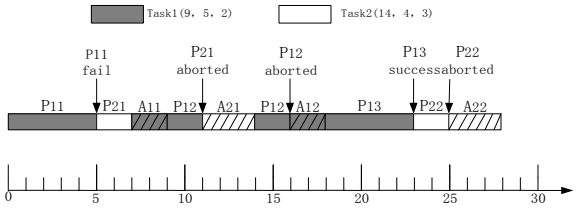

图 5 基本的主/副版本容错调度算法执行情况

5.3 改进的主/副版本容错调度算法

为此, 本文提出了一个改进型的主/副版本容错调度算法, 主要改进在于: 在处理器决定调度主版本任务前, 先计算其可用运行时间是否充足, 如果不充足, 就可以放弃该主版本任务, 减少处理器的无效执行时间.

假设在时刻 t, 主版本 $P_{i,j}$ 的最晚执行时间是 $NT_{i,j}$, $I_{m,n}$ 表示实时任务 $t_{m,n}$ 的副版本 $a_{m,n}$ 在任务 $t_{i,j}$ 的通知时间之前执行的部分, $AT_{i,j}$ 表示主任务 $P_{i,j}$ 的可用时间, P_i 是实时任务 T_i 的主版本任务的执行时间.

$$AT_{i,j}=NT_{i,j}-t-\sum_{t_{m,n}\in L} I_{m,n}$$

$$L=\{t_{m,n}\mid NT_{m,n}<NT_{i,j}\}$$

比较 $AT_{i,j}$ 和 P_i 的大小, 如果 $AT_{i,j}>P_i$, 则主版本任务 $P_{i,j}$ 可调度; 否则, 主版本任务 $P_{i,j}$ 不可调度. 可以看出, 通过加入主版本任务的可用时间的计算, 减少了肯定不能执行完成的主版本的时间, 提高了主版本任务的执行成功率.

如图 5 的例子, 在时刻 9, 处理器调度主版本之前, 先计算其可用运行时间, 易算出等于 4, 不满足其执行时间 5 的要求, 此时我们不选择运行 P12, 因为其最后只能执行时间较少的副版本, 转而继续执行 P21. 使用改进的主/副版本容错调度算法的调度执行情况如图 6 所示.

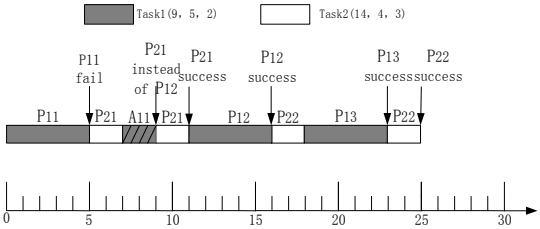

图 6 改进的主/副版本容错调度算法执行情况

从图 6 可以看出, 通过加入主版本可用时间的计

划, 减少了处理器的无效执行时间, 并且让后面更多的主版本成功执行, 获得了更好的计算质量.

算法伪代码描述如下:

```
Algorithm( )
Static:
alternateList: 用 backwards-RM 算法构造 alternateList, 随着时间升序排列;
primartList: 主版本任务列表, 随任务优先级降序排列;
t:当前时间,初始化为 0;
availbleTime:当前主版本的可用运行时间
candidate,具有最高优先级的就绪主版本;
While(TRUE)
    If t≥alternateList.head.notificationTime then
    执行 alternateList.head, 执行完后将它从表头移除;
      goto next;
      end
      for each primary Pij in primartList
      计算 availbleTime;
        If availbleTime≥Pij then
          把 Pij 设置为 candidate; break;
        end
      end
if candidate ≠ null then
  执行 candidate;
If candidate 执行成功 then
    把相应的副版本从 alternateList 中移除;
goto next;
next:
     设置下一个触发时间 t;
end
```

6 实验环境及结果分析

本文主要测试系统在模拟 SEU 干扰下, 容错机制对任务执行的成功率以及性能的影响.

故障注入[15]是模拟 SEU 的手段之一,MiniCore 系统提供一个故障注入服务, 能够向内存中的地址随机注入错误, 模拟空间环境中 SEU 的效果. 针对两级容错策略机制, 分别在系统级和任务级对系统的性能进行测试.

6.1 系统级容错测试

加入系统级容错策略后, 通过周期性的内存数据检错纠错, 使系统内存关键数据的准确性得到了保证. 系统级容错测试主要测试系统在未加入系统级容错和加入系统级容错后, 内核代码空间的大小变化.

测试平台: Ubantu 14.0.4+QEMU

测试参数: bzimage(MiniCore 内核镜像大小) cimiger(目标文件大小).text(代码段).data(数据段). Bss(静态内存分配段)

测试方法: ①在 MiniCore 源文件下使用 ls -l 命令查看目标文件和内核镜像大小; ②使用 readelf–a cimiger 查看目标文件中各个代码段的大小. 测试结果如表 1 所示.

表 1　程序代码的各个段大小

文件名	text	data	Bss	bzimage	cimiger
excutable.elf	18071	17703	49252	29675	114280
excutableFT.elf	19863	24310	65440	38530	151680

从表 1 可以看出, 在为操作系统加入系统级容错特性后, 最后生成的程序大小比不带容错的程序大小增加了大约 33%. 相对于其它的容错方法, 这个比例相对还是比较低的.

6.2 任务级容错测试

在 MiniCore 系统上定义 4 个周期性任务, 其任务集为 $\Gamma=\{\tau_1,\tau_2,\tau_3,\tau_4\}$,$(T_i,p_i,a_i)$=(14,3,2), (22,6,3), (28,6,4), (121,23,7), 那么单个计划周期为 3388 个时钟数. 分别用基本的主/副版本容错调度算法和改进的主/副版本容错调度算法来执行 10 个计划周期, 比较他们在主版本上的执行成功率. 测试结果如图 7 所示.

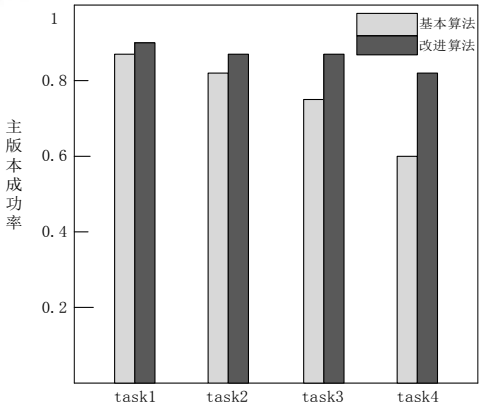

图 7　主版本任务成功率

从图 7 可以看出, 改进型的主/副版本容错调度算

法相比基本的主/副版本容错调度算法有更好的调度质量, 相对来说, 可以完成更多的主版本任务.

6.3 系统时间性能测试

通过比较不带容错机制的 MiniCore 系统和带容错机制的 MiniCore 系统在消息发送时间、中断处理时间和任务切换时间上的开销, 来验证两级容错机制对系统实时性的影响. 测试结果如表 2 所示.

表 2 MiniCore 带容错和不带容错的时间性能比较

性能指标(us)	未容错	系统级+任务级容错
消息发送时间	7	10
中断处理时间	10	24
任务切换时间	50	60

通过表 2 可以看出, 使用两级容错方案后, 对系统实时性还是有较小的影响, 但是, MiniCore 本身在时间性能上就有优势, 牺牲一部分时间来换取系统的可靠性提高是值得的.

6.4 对比分析

传统的软件容错技术中, 如 N 版本技术需要在内存中保存多个软件版本的代码, 且运行时空间开销大; 多个版本并发执行, 对系统实时性有较大影响; 功能实现中, 需要多任务管理、并发调度、结果表决等功能模块的支持.

相对而言, 本文的改进型主副版本技术在空间开销和时间开销上得到了合理控制, 功能实现部分只需增加主副版本容错调度和错误检测等模块, 较易实现. 能够适用于飞行控制系统、工业控制等领域.

7 结语

空间系统有高可靠性的要求. 为了减少 SEU 对系统内存数据的影响, 本文在系统级提出了基于周期性的内存检错纠错机制; 同时为了保证实时系统中任务的可靠运行, 本文在任务级提出了一种改进型的主/副版本容错调度算法; 通过两级容错机制, 花费约 33% 的空间开销和少量的时间代价, 能够在不影响系统实时性的情况下, 提升系统的容错能力. 相对于 N 版本技术等软件容错策略, 系统在时间和空间上的开销还是可以接受的. 以后需要进一步针对 MiniCore 系统的特点, 采用更多有效的软件容错手段, 在多 CPU 并行通信容错管理、恶劣环境下的软件容错支持等方面进行深入研究, 更好地提高系统的可靠性.

参考文献

1 Sprunt B, Sha L, Lehoczky J. Aperiodic task scheduling for hard-real-time systems. Real-Time Systems, 1989, 1(1): 27–60.

2 Laprie J. Dependable computing and fault-yolerant systems. Depndability: Basic Concepts and Terminology in English, French, German, Italian and Japanese, 1992, 5.

3 Persya AC, Nair TR. Fault tolerant real time systems. arXiv preprint arXiv:1001.3756, 2010.

4 Avizienis A. The n-version approach to fault-tolerant software. IEEE Trans. on Software Engineering, 1985, 11(12): 1491– 1501.

5 Allerton DJ, Jia H. Redundant multi-mode filter for a navigation system. IEEE Trans. on Aerospace and Electronic Systems, 2007, 43(1): 371–391.

6 Randell B, Xu J. The evolution of the recovery block concept. Software Fault Tolerance, 1995, 3:1–22.

7 罗威,阳富民,庞丽萍,李俊.基于延迟主动副版本的分布式实时容错调度算法.计算机研究与发展,2007,44(3).

8 李宏,陈香兰,吴明桥,等.服务体模型与操作系统内核设计技术.计算机研究与发展,2005,42(7):1272–1276.

9 吴明桥,陈香兰,张晔,等.一种基于服务体/执行流的新型操作系统构造模型.中国科学技术大学学报,2006,36(2): 230–236.

10 Mahrenholz D, Spinczyk O, Schroder-Preikschat W. Program instrumentation for debugging and monitoring with Aspect C++. Proc. Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002. (ISORC 2002). IEEE. 2002. 249–256.

11 Nelson VP. Fault-tolerant computing: Fundamental concepts. Computer, 1990, 23(7): 19–25.

12 Liestman AL, Campbell RH. A fault-tolerant scheduling problem. IEEE Trans. on Software Engineering, 1986, (11): 1089–1095.

13 Lengliz I, Kamoun F. A rate-based flow control method for ABR service in ATM networks. Computer Networks, 2000, 34(1): 129–138.

14 Chetto H, Chetto M. Some results of the earliest deadline scheduling algorithm. IEEE Trans. on Software Engineering, 1989, 15(10): 1261–1269.

15 孙峻朝,王建莹.容错机制测评中的故障注入模型及应用算法.计算机研究与发展,1999,36(11):1335–1341.