基于双数据通路的快速上下文切换方法①

权彦清[1], 陈香兰[1,2]

[1](中国科学技术大学 计算机科学与技术学院, 合肥 230026)
[2](中国科学技术大学 苏州研究院, 苏州 215123)

摘 要: 嵌入式实时操作系统对时间性能有着严格的要求. 上下文切换在实时操作系统中频繁发生, 其时间开销直接影响整个系统的实时性能. 针对一款拥有双数据通路、对存储系统具有并行访问能力的 DSP 系统, 研究出一种快速上下文切换的方法. 该方法将任务上下文相关的寄存器组分为两部分, 分别保存在可以并行访问的内存中, 通过 DSP 的双数据通路并行存取这两部分的内容. 该方法在一款开放源代码的操作系统 RTEMS 中进行了验证, 实验表明, 在该 DSP 系统中, 采用基于双数据通路的上下文切换方法能将上下文的保存和恢复时间降低为单数据通路的 49.04%.

关键词: 上下文切换; 实时性; 操作系统; DSP ; 双数据通路

Fast Context Switch Based on Dual Data Paths

QUAN Yan-Qing[1], CHEN Xiang-Lan[1,2]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)
[2](Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123, China)

Abstract: Time overhead is important in real-time operating system. Context switch is one major delay because it occurs frequently in a RTOS. The given DSP has features for having a dual data paths, providing multiple storage blocks can be accessed in parallel, general-purpose register banks can be accessed in parallel. To reduce the overhead of context switching, we proposed a solution based on the features of the given DSP. We split registers of context into two parts, and saved them in two memory parts which can be subjected to parallel access. When context switching, saving and restoring the registers of context via dual data paths is rather than one. This solution had been implemented in the RTEMS operating system. The experimental results show that saving and restoring the registers via dual data paths can reduce the processing time by 49.04% comparing to one data path.

Key words: context switch; real-time; operating system; DSP; dual data paths

1 引言

1.1 实时系统

近年来, 嵌入式系统在个人消费类电子产品、工业控制、通信以及国防领域使用地越来越广泛. 嵌入式系统主要由嵌入式处理器、相关支撑硬件、嵌入式操作系统及应用软件系统等组成, 它是集软硬件于一体的可独立工作的“器件”[1]. 实时操作系统(Real-Time Operating System, RTOS)以其具有多任务管理能力、可剪裁性、低功耗、高实时性等优点在嵌入式系统中得到广泛应用.

在嵌入式实时系统中, 对于系统实时性能的度量, 有以下基本标准[2]:

1) 有界的响应时间

2) 快速的响应时间

有界的响应时间是指能够提前确定代码的最坏执行时间(Worst-case Execution Time, WCET)[2]. WCET 是

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);江苏省自然科学基金(SBK2012194)

收稿时间:2015-03-09;收到修改稿时间:2015-04-26

确保实时系统可靠性的一种重要度量指标[3]. 对于上下文切换代码来说, 在确定的硬件平台上, 其代码的执行时间是能够提前确定的.

快速的响应时间是指响应时间能够尽可能的短. 上下文切换操作在系统中, 每秒会发生几十只几百次, 其带来的时间开销不可忽略[4]. 如果一个实时任务要求的时间限制小于上下文切换带来的时间开销, 那么该任务就无法在该系统上被执行[5]. 因此, 为了降低实时操作系统的运行时开销, 并使系统运行地更快, 实时操作系统应该有一个快速的上下文切换机制[6,7].

1.2 任务上下文

任务上下文是指任务在运行的任一时刻, CPU 内各种寄存器(例如通用寄存器, 程序计数器, 程序状态字等)的值.

对于可抢占的实时操作系统, 一旦高优先级任务就绪, 就有必要发生任务切换. 在任务切换时, 由操作系统完成任务上下文切换工作: 将正在执行的任务这一时刻的上下文内容保存到内存中某个位置中以备将来恢复; 然后从内存中取得将要执行的任务的上下文内容, 存储到相关寄存器中去, 使其恢复执行[4]. 此外, 在发生硬件中断时, 操作系统要保存被中断任务的上下文; 退出中断时, 内核要恢复被中断任务上下文. 由此可见, 上下文保存和恢复在实时操作系统中频繁发生.

在上下文切换中, 需要保存的寄存器数量和硬件平台密切相关, 且从几个到几千个不等. 通过对实时多处理器操作系统(Real-Time Executive for Multiprocessor System, RTEMS)[8]源码的分析, 在不考虑多处理器和浮点模块的前提下, x86 平台上需要保存的上下文数量是 24 个字节; SPARC 平台需要保存的数量是 88 个字节; 在 ARM 平台上则是 4 至 52 个字节. 专用于数字信号处理的数字信号处理器(Digital Signal Processor, DSP), 为了提升大量数据运算的性能, 系统中的寄存器数量高达上千个. 这些寄存器在上下文切换时都需要一一保存和恢复. 因此, 在 DSP 系统上, 上下文中寄存器切换的时间开销是不容忽视的.

目前, 提高上下文切换速度的方法有很多, 如增加寄存器有效位, 有选择地保存或恢复上下文[9]; 采用比硬盘访问速度更快的闪存介质, 并精心设计以凸显闪存的速度优势[10,11]; 采用影子寄存器, 避免任务切换时上下文的保存和恢复[12]. 上述上下文切换的优化方案大多是利用软硬件协同设计的方式来实现的.

本项目所用的 DSP 系统用于雷达信息采集, 需要进行大量数字信号转换和计算, 该 DSP 中寄存器数量高达 1303 个, 运行时会产生大量上下文信息. 考虑到该 DSP 具有的双数据通路传输能力、多个物理上分离的能够并行访问的存储区间以及分布在 A-B 两面的通用寄存器能够并行访问的特性, 我们设计一种双数据通路上下文管理方案来降低上下文切换中寄存器保存和恢复的时间开销. 本文给出了该 DSP 中单数据通路、双数据通路上下文切换的时间开销模型, 并给出基于 RTEMS 操作系统的实现方案和结果分析.

实验结果表明, 对比原型操作系统中上下文单数据通路管理方案, 本文提出的方案能够将上下文中寄存器保存和恢复的时间开销降低为原来的 49.04%.

2 DSP中上下文切换的模型及其改进

2.1 DSP 设备特性

1) 本多路数据总线并行传输

本文所使用的 DSP 系统提供了多路数据总线并行传输的能力. 该 DSP 共有三路数据总线, 同一时刻能够执行两读一写或两写一读的并行传输操作.

2) 多存储区间并行访问

该 DSP 的地址空间在物理上分为 6 个能够并行访问的存储区间. 将数据存储于不同的存储区间中, 则能够在同一时刻并行访问.

3) 支持多路访问的通用寄存器组

该 DSP 将通用寄存器组设计为两部分——A 面和 B 面. 当被寻址的两个寄存器分别位于 A 面和 B 面时, 它们可以被并行访问.

2.2 单数据通路寄存器切换过程及其时间开销模型

本文所提到的上下文相关寄存器, 是指在上下文切换时必须保存的寄存器. 在该 DSP 中有以下 5 类:

①B 面通用寄存器;

②控制寄存器;

③标志寄存器;

④返回地址寄存器;

⑤中断地址寄存器.

其中, A-B 面通用寄存器可直接通过数据通路保存到内存中, 其他寄存器则需通过 A-B 面寄存器转存, 恢复时亦然, 下文中分别使用 RABs 和 RCs 表示 A-B 面寄存器和其他寄存器.

该 DSP 中上下文相关寄存器保存流程如下:

①保存 RABs;

②转存 RCs 到 RABs 中;

③再次保存 RABs;

④重复②③步骤, 直到 RCs 全部保存完.

该 DSP 中上下文恢复的流程类似保存流程, 但先恢复 RCs, 后恢复 RABs.

令 T_0 为访存指令的执行时间, 因该 DSP 无数据 Cache, 故 T_0 为常数. 令 Na、Nb、Nc 分别为 A 面通用寄存器、B 面通用寄存器和 RCs 的个数, 则单数据通路上下文保存或恢复的时间开销 T 为:

$$T = T_{ab} + T_c + T_{transfer} \tag{1}$$

其中, T_{ab} 和 T_c 分别为 RABs 和 RCs 保存或恢复的时间开销, $T_{transfer}$ 是转存 RCs 到 RABs 中的时间开销.

令 T_a 和 T_b 分别为 A、B 面寄存器保存或恢复的时间开销, 则:

$$\begin{aligned} T_{ab} &= T_a + T_b \\ &= Na \times T_0 + Nb \times T_0 \\ &= (Na + Nb) \times T_0 \end{aligned} \tag{2}$$

根据 DSP 手册, RCs 寄存器个数 Nc 大于 RABs 寄存器个数, 因此:

$$T_c = Q \times T_{ab} + R \times T_0$$

其中 Q 和 R 满足如下关系:

$$N_C = Q(N_a + N_b) + R$$

即有:

$$\begin{aligned} T_c &= Q \times (Na + Nb) \times T_0 + R \times T_0 \\ &= Nc \times T_0 \end{aligned} \tag{3}$$

由于 RCs 转存到 RABs 无需访存, 单数据通路上下文切换中已利用指令最大并行能力完成转存, 其时间开销 $T_{transfer}$ 已最优, 此处不再展开分析.

根据公式 1、2、3, 进一步有:

$$T = (Na + Nb + Nc) \times T_0 + T_{transfer} \tag{4}$$

公式 4 表明在该 DSP 中进行上下文相关的寄存器保存或恢复的时间开销与寄存器个数和访存指令执行时间 T_0 成正相关关系. 根据 DSP 手册, T_0 是常数, 寄存器个数有 1303 个, 因此 T 非常大, 有必要对该 DSP 的上下文切换方法进行优化.

2.3 基于双数据通路的寄存器切换及其时间模型

为了降低上下文切换的时间开销, 参考文献 9,10,11,12 中的方法要么减少需要保存或恢复的寄存器数量、要么通过改变硬件架构来提高访存性能. 而我们的项目要求针对确定的 DSP, 不能进行硬件上的改造; 并且该 DSP 中上下文切换时必须保存或恢复的寄存器数量也是确定的, 故现有方法不能满足我们的项目需求. 因此, 我们考虑利用该 DSP 系统的特性, 设计一套降低上下文切换时间开销的软件方法.

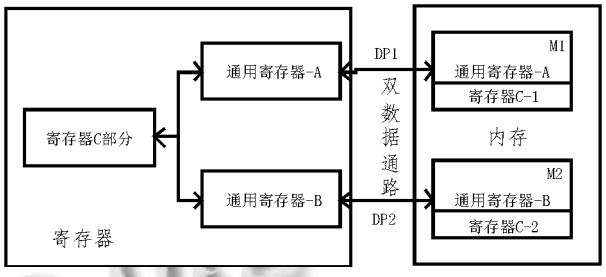

图 1　双数据通路切换方法设计图

根据 2.1 节特性 3, 对 RAs 和 RBs 能够并行访问; 对于 RCs 部分, 保存或恢复时, 使用 RAs 和 RBs 进行转存, 转存后, 也将通过 RAs 和 RBs 进行并行传输. 令 M1 表示 RAs 和利用 RAs 转存的 RCs 中的 C1 部分在内存中存储位置; 同理, M2 为 RBs 及相关的 C2 在内存中的存储位置. 根据 2.1 节特性 2, 将 M1, M2 分配在不同的内存存储区间, 从而能够对 M1, M2 进行并行访问. 如图 1, 在发生上下文切换时, 使用两条数据通路对以上两部分寄存器的值进行并行传输. 该方法相当于减少一条数据通路上需要传输的寄存器数量. 由公式 4 可知, 减少需要传输的寄存器个数能够实现降低系统中寄存器切换时间开销的目的.

在使用双数据通路的情况下, 上下文切换时间模型将满足以下关系:

保存或恢复 RAs 和 RBs 时间 T'_{ab}

$$T'_{ab} = \max(T_a, T_b) = \max(N_a, N_b) \times T_0 \tag{5}$$

保存或恢复 RCs 的总访存时间 T'_c

$$\begin{aligned} T' &= T'_{ab} + T'_c + T_{transfer} \\ &= (Q+1)(\max(Na, Nb))T_0 + \left\lceil \frac{R}{2} \right\rceil \times T_0 + T_{transfer} \end{aligned} \tag{6}$$

由公式 1,5,6 得出本方法中在上下文切换时寄存器保存或恢复的时间开销为 T', 有:

$$\begin{aligned} T' &= (Q+1) \times \frac{Na+Nb}{2} \times T_0 + \left\lceil \frac{R}{2} \right\rceil \times T_0 + T_{transfer} \\ &\approx \frac{Na+Nb+Nc}{2} \times T_0 + T_{transfer} \end{aligned}$$

因为 Na=Nb, 所以 $\max(Na, Nb) = Na = Nb = \frac{Na+Nb}{2}$ 故:

$$\begin{aligned} T' &= (Q+1) \times \frac{Na+Nb}{2} \times T_0 + \left\lceil \frac{R}{2} \right\rceil \times T_0 + T_{transfer} \\ &\approx \frac{Na+Nb+Nc}{2} \times T_0 + T_{transfer} \end{aligned} \tag{7}$$

令 Δ 为使用双数据通路进行寄存器保存或恢复能够节约的时间开销, 有:

$$\Delta = T - T' \approx \frac{1}{2} \times (Na + Nb + Nc) \times T_0 \quad (8)$$

由公式 4、7、8 可知, 节约的时间来自于减少了访存指令的总执行时间.

令 T_m 为单数据通路切换的时间开销 T 中访存指令的总执行时间. 有:

$$T_m = (N_a + N_b + Nc) \times T_0 \quad \frac{\Delta}{T_M} \approx \frac{1}{2} \quad (9)$$

由公式 9 可知, 使用双数据通路进行上下文切换, 能够减少将近 1/2 的访存时间.

$$\frac{T}{T'} = \frac{T_m + T_{transfer}}{T_m + T_{transfer} - \Delta} \approx \frac{T_m + T_{transfer}}{\frac{1}{2}T_m + T_{transfer}} \quad (10)$$

由公式 10 可知, 双数据通路切换方法的加速比取决于上下文切换代码中, 访存指令所用时间占整个上下文切换时间的比例.

我们将上述双数据通路上下文切换方法在 RTEMS 操作系统上实现, 对公式 9 的正确性进行验证和分析, 并根据公式 10 计算得出新方法的加速比.

3 基于RTEMS操作系统的实现方案

RTEMS 是一个专为嵌入式系统设计的开源实时操作系统. 目前, 在 RTOS 领域, 一类是 RTLinux, Windows-NT 等以现有的非实时操作系统为基础, 扩充了实时特性的操作系统; 另一类是 RTEMS, VxWorks[13]等专门针对实时应用场景设计的操作系统, 这类操作系统因其在设计时就考虑了实时性的标准, 因而能够更好的满足硬实时的要求[7,14].

作为一款源码开放的操作系统, RTEMS 易于进行源码级的修改和调试. 鉴于上述原因, 本文采用 RTEMS 操作系统作为原型对我们提出的双数据通路上下文切换方法进行实现和验证.

以下三节将从上下文存储位置的空间管理, 任务上下文初始化步骤以及上下文切换过程三个部分详述在 RTEMS 上双数据通路上下文切换方法的实现.

3.1 上下文存储空间布局与管理

本节给出上下文管理数据结构的定义, 并详述该数据结构使用的存储空间及其管理器的设计与实现.

3.1.1 数据结构定义

将保存于 M1 部分的上下文命名为上下文 A 部分, 保存于 M2 部分的上下文命名为上下文 B 部分.

单数据通路版本上下文管理数据结构定义如下:

```
typedef struct _Context_Control
{    REG_TYPE _ira[44];        //中断返回地址寄存器
     REG_TYPE _ra;             //程序返回地址寄存器
     REG_TYPE _ba;             //分支地址寄存器
     REG_TYPE _sr;             //程序计数器
     REG_TYPE _data[1256]; //其他上下文相关寄存器
}Context_Control;
```

上下文切换时需要保存的寄存器总数为 1303 个, 其中其他上下文相关寄存器 1256 个, 占总数的 96.54%. 其他上下文相关寄存器主要存储的是任务运行时产生的信息. 因此在划分两部分上下文内容的时候, 仅将该部分寄存器分为两个部分. 其中上下文 A 部分沿用单数据通路上下文管理数据结构, 上下文 B 部分仅包含一半的其他上下文相关寄存器.

1) 双数据通路上下文 A 部分数据结构定义

```
typedef struct _Context_Control
{REG_TYPE context_B_base; //指向B部分上下文基址
     REG_TYPE _ira[44];    //中断返回地址寄存器
     REG_TYPE _ra;         //程序返回地址寄存器
     REG_TYPE _ba;         //分支地址寄存器
     REG_TYPE _sr;         //程序计数器
     REG_TYPE _data[640];  //其他寄存器
}Context_Control;
```

2) 双数据通路上下文 B 部分数据结构定义

```
typedef struct _Context_B
{    REG_TYPE _data[616];  //其他寄存器
}Context_B;
```

对比单数据通路上下文数据结构, 上下文 A 部分多出一个用于存储 B 部分基址的成员, 而包含的其他寄存器的数量从 1256 下降为 640 个, 理论上, 上下文 A 部分存取时间将为单数据通路版本寄存器存取时间的 53.26%. 比较双数据通路上下文 A、B 部分数据结构, A 部分总数量为 702 个, B 部分总数量为 616 个. 其中 A 部分多出来的部分是管理数据所占据的空间.

3) 上下文 B 部分对任务管理的透明性

任务管理块中有管理该任务上下文的数据成员, 在双数据通路版本中, 为了避免上下文被划分成两部

分对任务管理的影响, 我们设计任务管理块中相关数据成员仅用来管理上下文 A 部分; 上下文 B 部分通过上下文 A 部分来管理, 从而使得任务不用感知到上下文 B 部分的存在, 进而不改变任务管理上下文的接口.

3.1.2 双 Workspace 管理器设计

大部分硬件平台上下文数量很少, 如 x86 平台仅有 24 个字节的信息, 所以 RTEMS 将上下文数据结构作为任务管理块数据结构(Task_Control_Structure)的一部分. 而该 DSP 上寄存器数量高达 1304 个字, 如果将上下文存放在任务管理块中, 存储需求太大; 并且上下文存储空间仅在任务被挂起时使用, 任务恢复运行后, 该空间存储的数据就无效了. 因此在本方案中, 对于单数据通路版本, 在任务挂起时, 将上下文保存在任务栈上, 在任务管理块结构中仅保存一个指针, 指向上下文在栈中的保存位置. 对于双数据通路版本, 上下文 A 部分沿用单数据通路管理方式, 保存在任务栈上; 上下文 B 部分则保存在可以和任务栈所在空间并行访问的存储区间中.

在 RTEMS 操作系统中内核使用的动态内存在 Workspace 管理的存储区间中分配, 上下文作为操作系统内核数据结构, 其存储空间在 Workspace 中分配. 为了使得在上下文切换的操作中对上下文 A、B 部分的操作方式保持一致, 上下文 B 部分所在存储区间也采用 Workspace 的管理方式, 且数据的存取顺序也采用栈的管理方式——FILO. RTEMS 现有的 Workspace 管理器是用来管理一块连续的内存空间的, 为了利用 DSP 分离的存储区间能够并行访问的特性, 我们需要修改 Workspace 管理器, 使其用来管理两个物理上分离的存储区间.

3.1.3 RTEMS 操作系统配置表

RTEMS 作为一款配置化操作系统, 用户需要在其配置表中指定 Workspace 管理的存储区间的起始地址和大小. 为了让 RTEMS 支持上下文双数据通路管理, 我们对配置表数据结构: rtems_configuration_table, 增加成员 work_space_B_start 指针, 它指向新增的 Workspace 管理的另一个存储空间的起始地址; 增加 work_space_B_size, 用来存储新增的存储区间的大小. 在初始化指定 Workspace 管理的两个存储区间起始地址时要保证它们位于可以并行访问的存储区间中.

3.1.4 双 Workspace 初始化过程

如图 2, 初始化 Workspace 管理器的工作由 BSP 任务完成. 此时处于操作系统初始化状态, 为 RTEMS 任务的运行准备环境. 根据 RTEMS 配置表中指定的 Workspace 管理的两个存储空间起始地址和大小, 调用 Workspace 初始化函数. 因为 Workspace 管理器是基于 Heap 管理方式的, 所以将指定的两部分内存空间初始化为由 Heap 方式管理的空间, 以备 RTEMS 操作系统后续为任务分配栈和上下文 B 部分空间使用. 由图 2 可知, 如果系统内存不能满足用户的配置要求, 系统启动就会失败; 如果对 Workspace 管理的任一存储区间初始化失败, 也会导致系统启动失败. 因为 RTEMS 中操作系统使用的内存从 Workspace 中分配, 如果 workspace 管理器初始化失败, 后续操作系统使用的内存就无法分配. RTEMS 操作系统将上述两种情况视为启动失败, 终止运行.

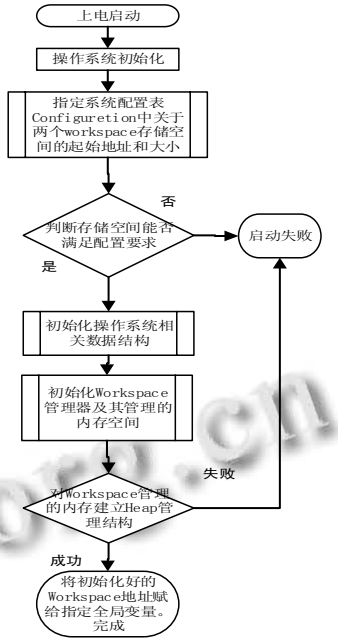

图 2 双 Workspace 管理器初始化流程图

3.2 任务上下文初始化

任务在第一次执行之前需要操作系统为其准备初始上下文. 本节描述为任务分配上下文 A、B 两部分存储空间的过程, 以及对上下文初始化的内容.

3.2.1 上下文 A 部分存储空间分配

上下文 A 部分位于任务栈中, 其空间的分配时机即任务栈开辟的时机. 对于 RTEMS 操作系统, 任务栈的分配在执行任务创建时进行.

如图 3, 在创建任务时, 首先为任务分配任务管理块数据结构, 它是操作系统中任务的标识, 保存了该

任务的相关信息. 然后为任务分配栈空间, 并将栈的位置信息存储在任务管理块中.

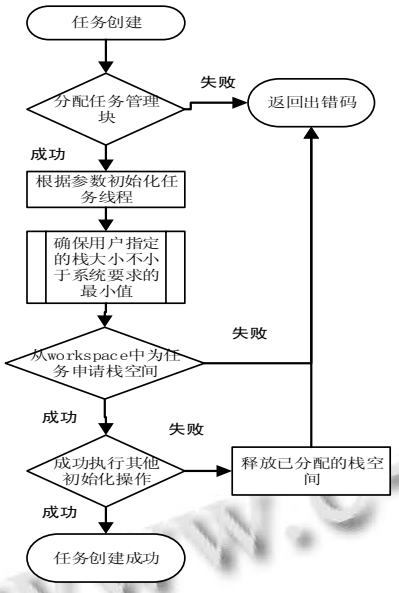

图 3 任务创建-上下文 A 空间分配

3.2.2 上下文 A 部分初始化时机和内容

RTEMS 中任务启动是指为任务准备除处理器以外的运行环境, 当任务被启动后, 一旦被调度到, 即可投入运行. 新任务被启动时, 需要加载运行环境, 其中包括了上下文的初始化操作. 图 4 是任务启动的流程.

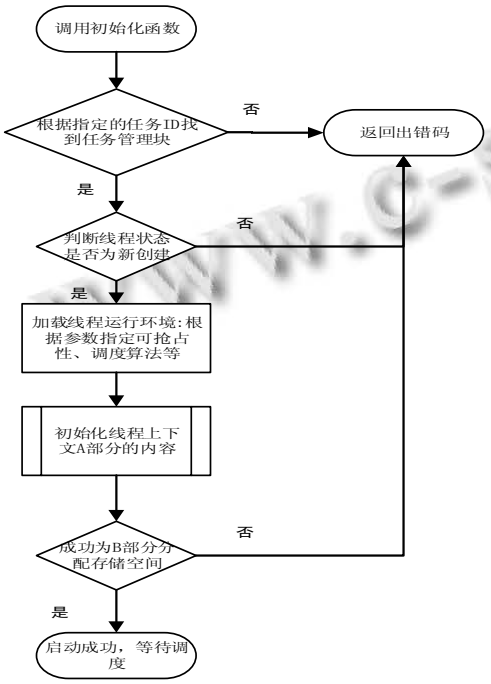

图 4 任务启动-上下文 A 部分初始化及 B 部分分配

上下文 A 部分初始化包括对以下几部分的初始化:

①将任务入口地址存入上下文 A 部分中指定位置

②B 部分空间的分配, 并将该地址存入 A 部分的对应位置

③确定上下文 A 部分在栈中的位置, 并将上下文 A 部分的起始地址存入任务控制块的指定位置

第一, 在 RTEMS 中, 所有任务在启动后跳转到统一的入口: _Thread_handler, 则将该函数入口地址作为返回地址存入上下文 A 部分中保存程序指针寄存器值的位置. 当新任务被调度到, 恢复上下文后, 即可从该位置开始执行, 这种设计统一了新任务投入运行和其他已运行过的任务恢复运行的操作.

第二, 由于上下文 B 部分由 A 部分进行管理, 所以, 为 B 部分分配空间的操作一直延迟到对 A 部分内容的初始化时. B 部分所在空间是 Workspace 管理器管理的另一部分物理空间, 同样基于 Heap 进行管理. 在 3.1.4 双 Workspace 初始化过程一节中, 我们已经为 B 部分所在的存储区间进行堆化, 此处, 直接调用 _Heap_Allocate 例程从已经初始化过的存储区间中为 B 部分分配内存. 将得到的内存地址加上 B 部分空间大小作为 B 部分基址存入 A 部分中指向 B 部分基址的指针成员中: 因为栈的管理方式是从高地指向地地址增长的, 所以将 B 部分的高地址作为 B 的基址.

第三, 任务需要通过任务控制块管理自己的上下文. 图 5 是初始化上下文保存在任务栈中位置的示意图, 在任务初始化后, 上下文 A 部分位于栈的最高地址处; 当任务被调度到, 恢复上下文后, 上下文所占的空间可以被任务作为自己的栈空间来使用, 此时已经到达栈空间的最高位置. 可见, 初始上下文所占用空间并不占用运行时栈的空间大小.

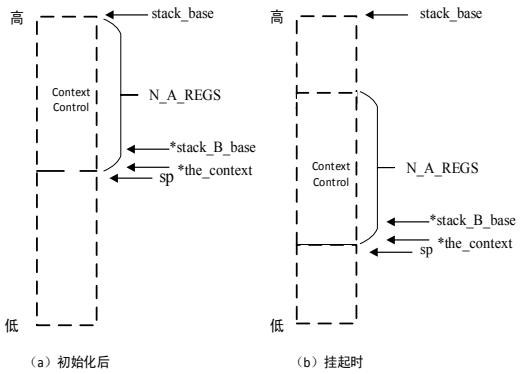

图 5 上下文 A 部分在栈中的示意图

3.2.3 上下文 B 部分零初始化开销

因为上下文 B 部分包含的寄存器中保存的是任务运行时产生的信息, 所以上下文 B 部分不需要进行初始化. 从而避免了增加上下文初始化的开销.

3.3 上下文切换

本节描述上下文切换中通用寄存器值的保存或恢复的过程. 本方案中上下文保存或恢复发生在以下四种场景中:

①BSP 任务到 RTEMS 任务

②任务间切换

③中断到任务的切换

④任务到中断的切换

在 1、2 的情况下, 会进行当前任务上下文的保存和下一个任务上下文的恢复. 3 仅进行被中断任务上下文的恢复, 4 进行被中断任务上下文的保存.

上下文中寄存器保存或恢复任务使用基于 DSP 系统的汇编代码实现. 根据上下文 A, B 部分数据结构的定义. A, B 部分各有一半的数据寄存器, 而所有的管理寄存器, 地址寄存器都存在 A 部分中. 在实现上下文保存的代码中, 以上下文 A 部分保存为主线, 在 A 部分开始保存数据寄存器时, 上下文 B 部分的保存开始进行, 并通过 DSP 汇编指令指定其与 A 部分保存指令并行执行. 当保存操作结束后, 将 B 部分的基址保存在 A 部分上下文的最后一个位置, 从而在恢复时, 能够从 A 部分中取得 B 部分上下文的恢复地址, 对于上下文恢复操作, 也类似保存操作的实现. 该方案实现了由上下文 A 部分管理 B 部分的设计.

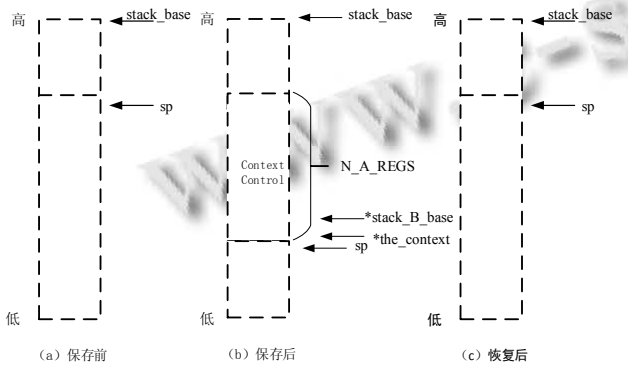

图 6　上下文 A 部分切换过程示意图

图 6 展示了上下文保存和恢复过程中, 同一个任务的上下文 A 部分在任务栈中的变化情况.

图(a)至(b), 是将上下文 A 部分保存在任务栈上的过程. 当正在执行的任务要被切换出去时, 将其上下文 A 部分从其任务栈当前栈顶位置开始保存, 保存数据寄存器时, B 部分并行执行.

图 6(b)至(c)是恢复任务上下文时 A 部分在任务栈中的过程. 如图 6(b), 从 A 部分保存在栈上的内容中取出 B 部分保存的地址, 当开始恢复数据寄存器时, A, B 两部分上下文并行执行.

任务恢复后, 如图 6(c), 此时的状态和发生切换前(如图 6(a))一样, 说明了将上下文保存在任务栈上并不影响任务栈运行时的空间大小和任务栈管理.

4 实验结果与分析

4.1 上下文双数据通路切换性能提升

本文提出的双数据通路上下文管理方案目的在于降低上下文切换中对寄存器组保存和恢复的时间开销. 该 DSP 系统提供了能够运行在 Windows 操作系统上的模拟器, 并且提供了图形化视图用来查看断点处相关寄存器和内存的值, 存在一个能够直接读取时钟周期数的寄存器. 以下实验运行在该 DSP 模拟器上.

在 DSP 模拟器上运行单数据通路和双数据通路版本的系统, 并在上下文切换开始和结束的位置设置断点, 因切换发生在关中断的时间内, 此区间内系统不会发生调度, 故记录的时间是确定的. 通过读取断点处所用的时钟周期数, 我们计算得到上下文切换所消耗的时间. 测量结果见表 1.

表 1　上下文切换的时钟周期数

	双通道	单通道	差值	比例(%)
通用寄存器切换时间	160	314	-154	49.04
整体上下文切换时间	690	837	-147	17.56

通过表 1 中的数据可知, 双数据通路的方案将寄存器保存和恢复的时间降低为单数据通路版本的 49.04%. 该结果和双数据通路上下文切换时间模型中公式 9 的结果一致, 说明在该 DSP 系统中, 本方法能够有效降低上下文切换时间开销. 在上下文切换中, 寄存器组访存指令所

用时钟周期数占整个上下文切换代码的 37.51%(314/837=37.51%), 根据公式 10, 计算得出双数据通路方法的加速比为 1.23.

第一行数据显示, 在每次发生任务切换时, 对比单数据通路的切换方案, 使用双数据通路切换方案能够减少 154 个时钟周期. 由 3.1.1 节中上下文 B 部分数

据结构 Context_B 定义可知, 该部分包含的寄存器个数为 616 个, 每个寄存器大小为一个字. 在我们的 DSP 中, 每个 cycle 内能够传输连续的 8 个字. 因此 616 个字, 用时 77 个 cycles. 在上下文切换中, 分为保存当前任务和恢复将要执行任务两个部分, 为此, 一共节约 77*2=154 个 cycles. 而整体上下文切换时间只减少 147 个 cycles 是因为在上下文双数据通路切换代码中增加上了 B 部分基址在 A 部分中保存和恢复的相关代码.

4.2 上下文双数据通路管理开销分析

为了实现双数据通路上下文切换的管理方案, 我们在 RTEMS 操作系统中添加或修改了相关的数据结构, 以及管理例程. 增加的管理开销见表 2.

表 2 上下文切换的额外管理开销(单位: 时钟周期数)

	双通道	单通道	差值	比例(%)
OS 初始化时间	171568	169146	2422	1.41
加载任务运行环境时间	23193	20281	2912	14.36

1) 存储 B 部分上下文的存储区间初始化的开销: 在 RTEMS 初始化时, 需要为 Workspace 管理的存储区间建立堆管理方式的数据结构. 因为 B 部分上下文存储的空间采用 Workspace 管理, 所以要对该存储区间进行堆化操作. 该开销在操作系统运行生命周期中只发生一次. 通过在 DSP 模拟器上运行结果看出, 对该存储区间的堆化操作所用时间只占整个操作系统初始化总时钟周期数的 1.412%, 具体数据见表 2 中第一行 OS 初始化时间. 如图 7 所示, 当操作系统中上下文切换的总次数达到 17 次, 由操作系统初始化时初始化上下文 B 部分所用存储区间的开销即可被抵消.

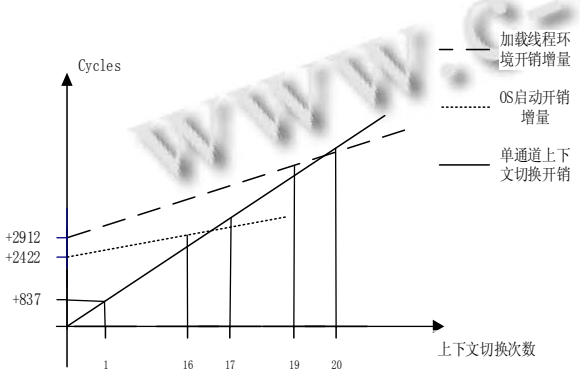

图 7 管理开销与上下文切换次数关系图

2) 在上下文 A 部分初始化时, 需要为上下文 B 部分分配存储空间, 该操作增加了任务启动中加载运行环境的开销. 数据见表 2 第二行. 该开销在一个任务的生命周期中发生一次, 而上下文切换发生在任务每次切换的过程中. 如图 7 所示, 当某个任务切换次数达到 20 次, 该开销即被抵消. 周期性任务是硬实时系统中主要的任务模型[15]. 比如雷达应用中的数据采集任务就是一种周期性任务, 该任务在每个周期内都要进行一次数据采集, 那么在多任务系统中, 该任务在每个周期内都将发生上下文保存和恢复. 因此, 本文提出的方案更适用于具有多个长期运行的周期性任务的实时操作系统.

5 总结

本文通过分析 DSP 上下文相关寄存器切换时间的模型, 利用该 DSP 系统的双数据通路, 存储系统具有并行访问能力的特性, 提出一种降低上下文相关寄存器切换时间开销的方法, 并给出基于 RTEMS 操作系统的具体实现方案. 新方案改进了 RTEMS 的 Workspace 管理器, 实现对两块能够并行访问的存储区间的管理. 借助改进的管理器, 我们将任务上下文相关的寄存器分为两部分, 分别保存在不同的存储区间内, 利用双通路数据通路, 实现上下文保存或恢复时两部分寄存器值的并行传输.

评价实时性能的另一个指标是有界的响应时间. 本文提出的方案, 从操作系统上下文管理的层面利用 DSP 系统的硬件特性降低了上下文切换的时间开销, 但切换时间长短还是取决于硬件平台所需要保存的寄存器数量, 所以新方案的时间性能依赖于具体的硬件设备. 因此, 未来将进一步考虑如何从操作系统管理层面出发, 保证上下文切换时间能够独立于硬件平台的寄存器数量.

参考文献

1 Robert I, Davis, Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. ACM Computing Surveys, 2011, 43(4): 1–44.

2 Masmano M, Ripoll I, Alfons Crespo, et al. TLSF: a new dynamic memory allocator for real-time systems. Proc. of Euromicro Conference on Real-Time Systems (ECRTS'04). 2004. 79–86.

3 Kopetz H. Real-time systems: design and principles for distributed embedded applications 2nd ed. Springer, 2011.

4 Context Switch Definition. http://www.linfo.org/context_switch.html. 2006 May.

5 Nader I, Rafla DG. Hardware implementation of context switching for hard real-time operating systems. Proc. of Midwest Symposium on Circuits and Systems. 2011. 1–4.

6 Hambarde P, Varma R, Jha S. The survey of real time operating system: RTOS. Proc. of Electronic Systems, Signal Processing and Computing Technologies (ICESC). IEEE. 2014. 34–39

7 John A, Stankovic, Rajkumar R. Real-time operating systems. Real-time System, 2004, 28(2-3): 237–253.

8 RTEMS home page. https://www.rtems.org/. 2014.

9 Zhou XR, Petrov P. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. Proc. of Design Automation Conference. 2006. 352–257.

10 Koh K, Lee SH, Yun S. Selective context switching scheme on flash memory. Proc. of International Conference on Computational Science and its Applications. 2009.

11 Flash memory. http://en.wikipedia. org/wiki/ flash_memory, 2015 January.

12 孙康,沈海斌,王继民,等.基于映像寄存器构建的实时操作系统内核.清华大学学报(自然科学版)2007,47(S2): 1899–1902.

13 VxWorks. http://en.wikipedia.org/wiki/vxworks. 2014-11.

14 Barbalace A, Luchetta A, Manduchi G, et al. Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application. IEEE Trans. on Nuclear Science, 2008, 55(1): 435–439.

15 涂刚.软实时系统任务调度算法研究[博士学位论文].武汉:华中科技大学,2004.