北京理工大學

本科生毕业设计（论文）外文翻译

外文原文题目：Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system

中文翻译题目：事件触发、实时操作系统中的并行、硬件支持的中断处理

在 QEMU 模拟器中的共享调度器设计与实现

Design and Implementation of a Shared Scheduler in QEMU Emulator

学　　院：计算机学院

专　　业：计算机科学与技术

班　　级：07112005 班

学生姓名：罗熙

学　　号：1120202534

指导教师：陆慧梅

事件触发、实时操作系统中的并行、硬件支持的中断处理

Fabian Scheler，Wanja Hofer，Benjamin Oechslein，Rudi Pfister，

Wolfgang Schröder-Preikschat，Daniel Lohmann

译者：罗熙

摘　要

事件触发的实时系统中的一个常见的问题，就是实现为中断处理程序的低优先级任务会中断实现为线程的高优先级任务。这个问题被称为单调速率优先级倒置，而当前基于软件的解决方案在更复杂的调度器功能方面受到限制，正如 AUTOSAR 嵌入式操作系统规范所要求的那样。

我们提出了一个基于硬件的方案，其可以利用一个协处理器来消除可能的优先级倒置。我们通过评估它的一个原型实现，展示我们的方案不仅克服了软件方案的限制，也用很小的处理开销换来了更好的可预测性。

关键词：CiAO 操作系统，实时系统，优先级驱动，单调速率优先级倒置，TriCore 微控制器，中断处理

第 1 章 引言

除了功能正确性之外，及时性时实时系统最重要的属性，因为无法按时交付的结果可能与错误的结果产生相同的影响。因此，提前进行可调度性分析，以确保所有带着硬性截止时间的任务可以按时完成，是必要的。系统整体的可预测性越好，这样的可调度性分析越可能得出精确和可靠的结果。

1.1 单调速率优先级倒置

不幸的是，现有的事件触发系统维护的优先级空间被分裂成两部分：硬件管理着中断的优先级，而操作系统管理着线程的优先级。Leyva-del-Foyo 等人说明，实时系统会因为这种二分的优先级空间受到严重影响[5]：与低优先级或者软实时的任务相关的中断，可能打断高优先级的硬实时任务。这会严重影响高优先级任务的响应时间，以至于它们可能无法在截止时间前完成。这种由实现为中断处理程序的低优先级任务引起的干扰被称为*单调速率优先级倒置（rate-monotonic priority inversion）*。对这样的软实时任务的出现频率，通常只能做非常弱的假设，这同时也让系统的行为更不可预测了。因此，可调度性分析的结果也会有更低的精确度，以及更悲观的结果。

为了解决这个问题，Leyva-del-Foyo 等人建议将所有任务实现为线程，并且用很简短的中断处理程序来触发它们，例如，通过设置信号量[5, 6]。更进一步地，在软件中实现的中断分级机制能通过屏蔽低优先级的中断源来杜绝这些中断处理程序的干扰。然而，这样的方案不适合需要更复杂的调度器功能，比如存储一个程序的多个激活实例或者允许多个任务使用相同的优先级，的系统。例如，被汽车微控制器广泛使用的 AUTOSAR-OS 标准[2]，就规定了这些功能。

1.2 并行、硬件支持的中断处理

我们的方法*基于硬件*，并且使用不屏蔽任何中断的方法阻止了不必要的打断。相反，我们将中断请求重定向到协处理器，让它们与正常的程序并行处理。我们的方案改进了 Leyva-del-Foyo 等人的方案，解决了其中的缺陷：

- 可以同时储存*任务的多个激活实例*。而在基于软件的方案中，每个任务只有一个额外的激活实例可以被硬件缓存。
- 多个任务可以*共享相同的优先级*，同时保留它们被激活的顺序。基于软件的方案无法做到这一点。
- 事件发生时，我们不使用信号量通知任务。在基于软件的方案中，显式等待信号量使*栈共享技术*无法实现，而这一技术在嵌入式系统中广泛地用于节省紧张的内存资源。

第 2 章对我们的方法做了详细介绍，并说明了它相比基于软件的方法的改进。我们在 TriCore 微控制器上的 CiAO 操作系统中实现了我们方案的原型（见第 3 章），并评估了它的性能（见第 4 章）。第 5 章列出了我们的方法的相关工作。第 6 章讨论了我们方法的几个方面——例如我们方法的中断过载问题，以及它的通用性。

第 2 章 设计

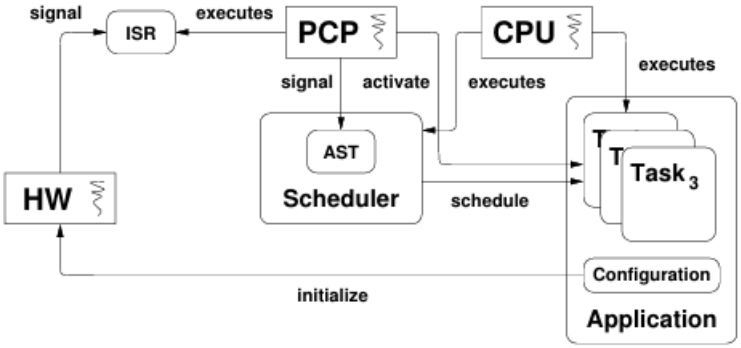


图 1 并行中断处理的设计

图 1 展示了我们设计的概要；它需要一个与主 CPU 并行工作的额外硬件。该硬件需要处理其它外部硬件提出的中断请求、访问调度器数据结构，以及向 CPU 发送一种信号——例如，通过处理器间中断。因为我们实现的平台是 TriCore 微控制器，我们使用外部设备控制处理器（peripheral control processor，PCP）作为该硬件，不过任何具有相似属性的硬件都可以用于该用途。

我们消除单调速率优先级倒置的方法是将所有硬件中断请求重定向到 PCP，并在 PCP 上处理它们。之后，我们配置中断控制系统，构建从任务优先级到中断优先级的映射，从而构造统一的优先级空间，而统一的优先级空间是避免单调速率优先级倒置所必须的。支持这些功能的中断控制系统在当前技术水平的，用于嵌入式系统的微控制器中很常见，TriCore 就是其中之一。映射自身是依靠程序完成的，只需要合适地初始化中断系统即可。

PCP 协处理器上的中断处理有责任唤醒与给定中断源相关的任务。为了进行这一唤醒操作，PCP 需要访问调度器的数据结构。当且仅当被唤醒的任务的优先级是系统的就绪任务中最高的，PCP 才会通知 CPU 需要调度一个不同的线程。该通知是通过向 CPU 发起一个*异步系统陷入（asynchronous system trap，AST）*实现的。AST 被实现为 *CPU 上的*中断处理程序，功能仅是触发 CPU 的重新调度。与在 PCP 上执行的 ISR（interrupt service routine，中断服务程序）不同，AST 只会在更高优先级的任务就绪时触发，触发时只会通知 CPU。因此，它自身不会引起单调速率优先级倒置。

2.1 PCP 中断处理程序的设计

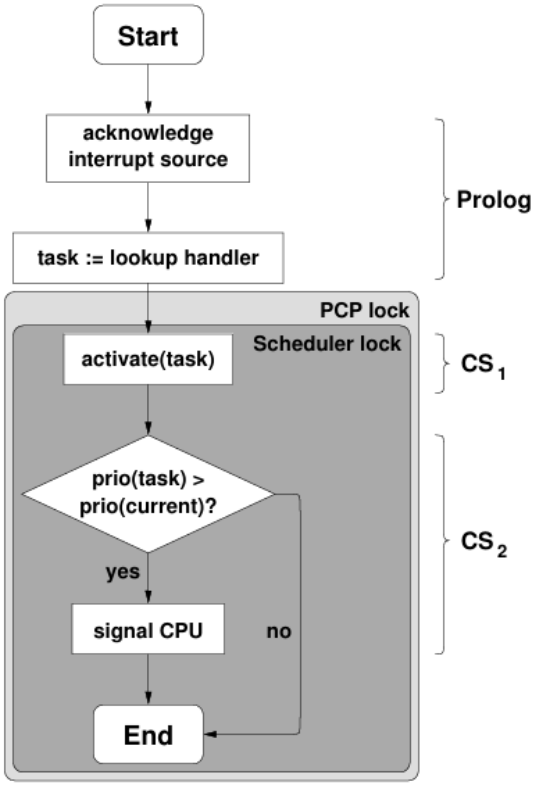


图 2 PCP 中断处理程序的结构

当每个中断请求发往 PCP 时，执行的中断处理程序的控制流结构如图 2 所示。

首先，确认发出当前请求的中断源。之后，从程序配置中查找并唤醒与这个中断源相关的任务。最后，将当前运行任务的优先级与刚才唤醒的任务的优先级比较，*在必要时*使用处理器间中断通知 CPU 进行重新调度。

为了让唤醒高优先级任务的开销尽量小，PCP 上的 ISR 可以被高优先级的中断请求打断。这与 PCP 上的 ISR 和 CPU 上的任务*并行*执行都表明了对常用数据结构的同步访问的必要性。通过保证两个临界区 CS1 和 CS2 的互斥访问（见图 2）达到同步的效果。这些临界区被两个嵌套的锁管理。外层的 PCP 锁通过锁定 PCP 上的中断以防止 ISR 被高优先级的中断请求打断。内部的锁通过 Peterson 的算法，一个基于自旋锁的互斥访问算法[9]，保证对调度器的互斥访问。PCP 锁需要在调度器锁之前获取；否则，如果 ISR 试图获得被另一被打断的 ISR 已经获得的调度器锁，就会引起死锁。

我们决定连续，而非分离地保证 CS1、CS2 两个临界区。因为我们在内核和中断处理函数的同步中使用了 Peterson 的算法，中断处理程序每次试图获得调度器锁时都可能被内核阻塞。此时引入的阻塞时延可能比 CS1、CS2 两个临界区共同的处理时间还长。（同时见 4.3 章）

ISR 的开始（prolog）阶段不需要同步，因为我们实现的平台是静态配置系统，例如 AUTOSAR OS[2]或 OSEK OS[17]；即是说，中断源的连接不会在运行时改变。

然而，将中断请求关联的任务唤醒（CS1 临界区）显然需要同步。它需要保证不被其它中断源的请求打断，因为它们同样会唤醒任务，因而会操作相同的数据结构。唤醒操作同样需要与来自 CPU 的交错访问同步，因为 CPU 上执行的任务也能以同步的方式修改调度器数据结构——例如，当任务结束时。

第二个临界区（CS2）需要同步，原因是防止 CPU 上的单调速率优先级倒置。假设当前 ISR 在检查是否需要重新调度之后、告知 CPU 需要重新调度的结果之前被打断，那么主动打断的高优先级 ISR 与被打断的低优先级 ISR 都会发出中断。这会导致 CPU 接受到的一个中断不会带来任务切换，因为 CPU 已经重新调度、切换到了高优先级的任务。额外的调度器锁也是必要的，因为 CPU 上的任务可能改变系统状态，使得重新调度不再必要——例如，通过激活和调度一个最高优先级的任务。这个情形也会导致 CPU 接收到一次不必要的中断，其可以被理解为单调速率优先级倒置。

由此，PCP 中断处理程序的设计杜绝了单调速率优先级倒置，同时尽可能减少了中断时延。

2.2 相较基于软件的设计的优势

与避免单调速率优先级倒置的基于软件的方法（例如在[5]中提出的）相比，并行中断处理的方法不需要屏蔽低优先级的中断。这是因为本方法中的任务分级是由硬件的中断系统，而非软件实现的。而硬件的中断系统本身就是为该目的而设计的。

我们的方法因此带来了一些显著的概念优势。

2.2.1 同一任务的多个激活实例

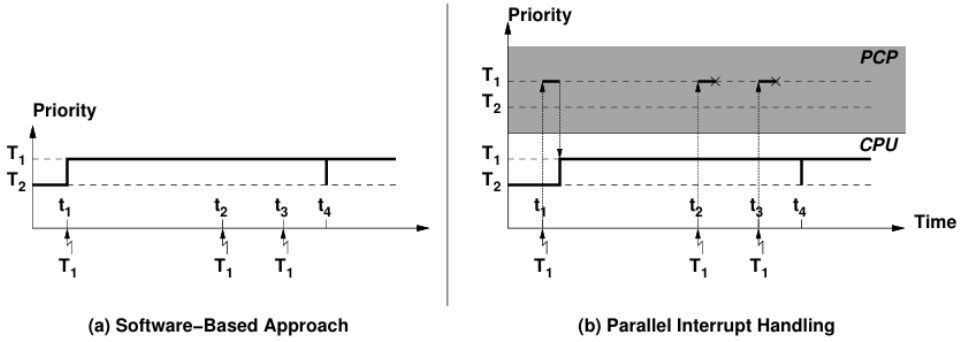


图 3 多个激活实例的例子的运行时行为

类似[5]的行为，屏蔽或取消屏蔽中断源以避免单调速率优先级倒置，导致了在处理被唤醒不止一次的任务时的不灵活，正如图 3（a）的例子。图中展示了两个任务 T1 和 T2，T1 的优先级高于 T2。在 t1 时刻，任务 T1 被第一次唤醒，也因此抢占了真正运行的任务 T2。在任务 T1 执行过程中，其又分别在 t2 和 t3 时刻各被唤醒了一次。在 t2 时刻的第一次唤醒可以被中断系统储存（假设使用了等级触发的中断系

统），但 t3 时刻的第二次唤醒一定会丢失。因此，当 T1 在 t4 时刻终止时，它只会被重新执行一次。

我们的方法不会被这一限制影响，就像图 3（b）中对同一例子演示的那样。因为没有中断源被屏蔽，t2 和 t3 时刻的中断请求可以被 PCP 服务，使任务 T1 被唤醒两次，并在 t4 时刻后额外执行两次。因此，例如 AUTOSAR-OS 标准要求的那样，任意多次唤醒同一任务的情形，只要调度器支持就可以处理。

2.2.2 同一优先级的多个任务

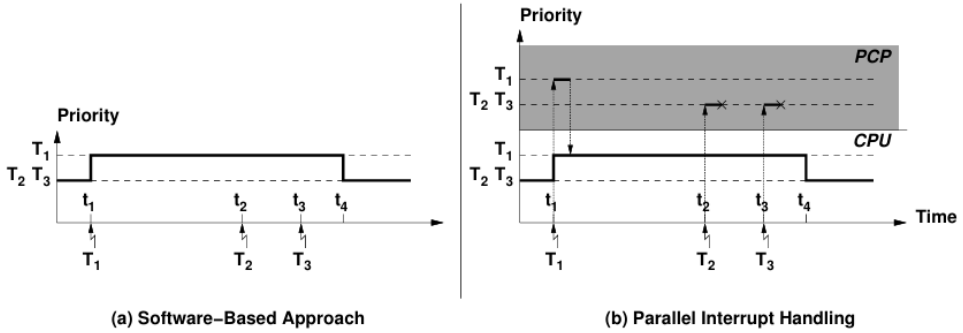


图 4 共享优先级的例子的运行时行为

基于软件的方法在处理多个任务共享同一优先级的情况同样能力有限，正如图 4（a）的演示。在 t2 和 t3 时刻，两个相同优先级的低优先级任务 T2 和 T3 被唤醒；中断请求被硬件缓存，因为那时中断源被屏蔽了。因此，在任务 T1 在 t4 时刻终止时，T2 和 T3 谁先执行只由硬件决定。

而我们的方法则保留了任务唤醒的顺序，正如图 4（b）的演示。因为中断未被屏蔽，任务可以按照它们出现的顺序唤醒，也按照这个顺序放入调度器。因此，例如 AUTOSAR-OS 标准要求的那样，共享同一优先级的不同任务可以按照它们到来的顺序被处理。

2.2.3 栈共享

基于软件的设计不仅限制了调度器功能，也阻碍了栈共享技术，一个在嵌入式系统中广泛地用于节省紧张的内存资源的技术，的实施。软件的分级要求所有事件处理函数都是被调度器管理的线程，并在事件发生时通过单侧同步机制，例如被硬件上运行的 ISR 发出的信号量告知。因此，每个线程需要等待一个相应的信号量，这种阻塞式、单侧的同步让栈共享的实施变得低效[12]。

我们的方法不需要阻塞式的操作系统原语，不对栈共享产生限制，也因此优化了系统的内存使用。

第 3 章 实现

我们在 TriCore 微控制器平台上，通过扩展 CiAO 研究操作系统，实现了我们方案的原型。在接下来的小节中，我们会描述操作系统的环境、硬件平台的相关特点，以及中断处理程序实现的细节。

3.1 CiAO 系统

CiAO（CiAO is Aspect-Oriented，CiAO 是面向方面的）是一类可配置的操作系统，它们用于嵌入式和深嵌系统，支持将在下文详细描述的 TriCore 硬件平台[14]。我们用于扩展和评估的指定系统源于 OSEK-OS[17]和 AUTOSAR-OS[2]标准。它因此实现了一个事件触发的操作系统，其使用带有多级队列、没有时间片、静态分配线程优先级的调度器。

在基本配置中，适用于 TriCore 平台的 CiAO 内核通过在进入内核时，将中断优先级提高到最高优先级的 ISR 的优先级，退出内核时将其重置为 0 的方式，实现内核与 ISR 的同步。通过这种方式，这段时间内的 ISR 触发会被推迟，直到退出临界区。在本文所述的扩展中，执行任务的 CPU 只会被一种中断打断：由 PCP 触发的异步系统陷入（AST）。因此，内核通过在系统调用期间，将中断优先级设置为 AST 的优先级的方式，与 AST 进行同步。

因为 CiAO 使用面向方面编程（aspect-oriented programming，AOP[10]）的技术实现——即 AOP 语言和 AspectC++织入器[19]——对内核的修改实现为一个单独的、封装好的方面模块。该方面包括将内核与 AST（如上文所述）和 PCP（使用 Peterson 的算法，见 2.1 章）同步的代码。使用*量化*这一 AOP 功能，对系统的横切交错的修改（每个系统调用都需要配备同步代码）可以封装在精简和因此可维护的代码中。

3.2 TriCore 平台

我们的原型在 TriCore TC1796 微控制器上实现，正如之前提到的，该硬件具有实现我们的设计需要的所有功能。TriCore 的外部设备控制处理器（peripheral control processor，PCP）提供一个与主处理器并行执行的线程，可以用于中断处理。在 TriCore 架构中，中断源用服务请求节点（service request nodes，SRNs）表示。每个 SRN 可以分配给一个服务提供者（CPU 或者 PCP），并且指定一个中断等级。因此，SRN 可以被静态配置为直接使用硬件实现中断分级方案，正如第 2 节的描述。PCP 和主 CPU 间的数据交换通过在特殊的总线间共享的内存完成；该内存在两个处理器中被映射到不同的位置。

我们配置该系统，从而让所有外部中断发往 PCP 而非主处理器。在 PCP 上运行、直接响应中断的程序按照我们在 2.1 章的设计实现，其会在接下来的小结详细描述。唯一仍然发往 CPU 的中断用于触发 AST。该机制由 PCP 使用，在比正在运行的任务优先级更高的任务被激活时，触发主处理器上的重新调度。

为了将被调度器管理的线程、中断源、被 PCP 管理的 ISR 相联系，我们使用一个存储在共享内存中的表格，将线程映射到中断源。该表格在系统启动时初始化。

3.3 PCP 中断处理程序的实现

因为多中断源的确认由 TriCore 平台的硬件自动完成，开始（prolog）阶段只包括获取和修改操作系统中的数据结构前的必要准备工作。它们包括查询静态数据，例如当前中断请求的所属线程的优先级，或者内核调度器数据结构，例如线程的就绪列表的地址。因为这些都是对静态数据的只读操作，这一部分可以被更高优先级的 PCP 中断处理程序或者 TriCore CPU 上运行的主程序流打断。

在第一个临界区 CS1 期间，处理程序将与当前中断源相关的线程加入调度器的就绪队列中。这部分使用了开始阶段查询到的信息。

在第二个临界区 CS2 中，如果重新调度是必要的，PCP 需要向 CPU 发送信号。为了确定是否是这种情况，它需要检查新唤醒的线程是否有比正在执行的线程更高的优先级。除此之外，它还要确认就绪队列中没有更高优先级的线程，因为如果更高优先级的线程在就绪队列中，它会马上被调度运行——将系统的优先级提高到它的优先级，使 AST 不再必要。如果这两种情况都不满足，我们的线程事实上优先级最高，则 PCP 向主处理器发送 AST 以触发重新调度。这一设计通过将是否发送 AST 的决定和发送 AST 的过程都原子化，确保每个发送的 AST 都一定会引起一次主处理器的重新调度，因为在 AST 触发前，没有其它线程会被唤醒。这一性质因此杜绝了单调速率优先级倒置。

对临界区 CS1 和 CS2 的保证在 2.1 章中描述。它通过在执行流离开临界区前禁用中断，避免了更高优先级 PCP 程序的打断。使用 Peterson 的算法实现的自旋锁，避免 PCP 和 CPU 对调度器数据结构的同时访问。

第 4 章 实验评估

我们以两个目的详细地评估我们的原型：一方面，我们证明我们的原型杜绝了低优先级和软实时的 ISR 的*多余的*干扰，另一方面，我们量化了我们实现的统一优先级空间带来的开销。

4.1 实验方法

在我们的实验中，我们使用一块 Infineon 的 TriBoard TC1796，其具有 TriCore TC1796 微控制器。它的时钟频率为 50MHz，导致其中断周期为 20ns。我们使用 Lauterbach TRACE32 硬件调试器与追踪器来精确测量 CPU 的所有执行事件，但追踪功能无法用于 PCP。因此，为了测量 PCP，我们改变 I/O 引脚的电平并使用数字存储示波器来确定 I/O 引脚上斜率的持续时间。我们使用一个函数发生器产生不同频率的方波信号来模拟 TC1796 的 I/O 引脚上的外部事件。为了测量，我们独占使用 TC1796 的内部、无等待状态的 RAM，以存储代码和数据。

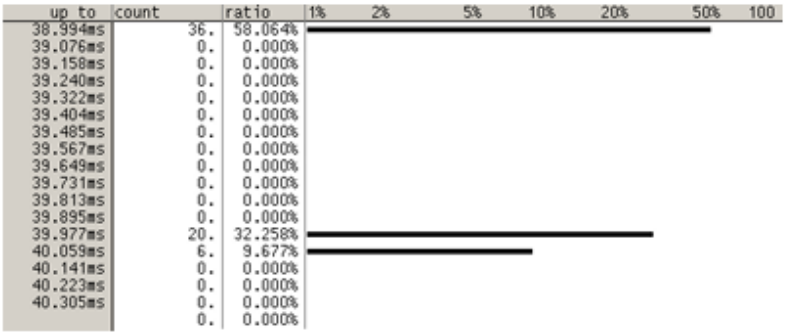
4.2 行为

我们第一部分的评估显示我们的原型的确可以防止单调速率优先级倒置为 CPU 带来的多余干扰。我们合成的任务集如表 1 所示；其包括一个硬实时任务 T1 和一个软实时，却可能干扰 T1 的任务 T2。在一个情景中，我们将软实时任务实现为传统的二分裂的优先级空间中的 ISR。在另一个情景中，我们将它实现为使用我们并行中断处理原型的线程。

我们测量了实时任务在操作系统具有二分裂的优先级空间的传统中断处理方法，以及在中断和任务具有统一优先级空间的我们的实现，这两种情形下的反应时间。我们通过检查 TRACE32 调试器生成的执行追踪，以及同样使用该调试器测量任务 T1 的运行时间，来确认我们的原型按预期工作。

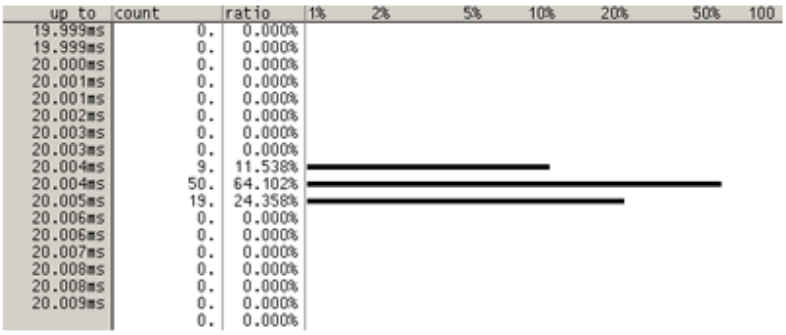
表 1 用于评估的样本任务

Task	Period	Deadline	Execution Time
T_1	50 ms	50 ms	20 ms
T_2	2 ms	– (soft)	1 ms



up to	count	ratio	1% 2% 5% 10% 20% 50% 100
38.994ms	36.	58.064%	
39.076ms	0.	0.000%	
39.158ms	0.	0.000%	
39.240ms	0.	0.000%	
39.322ms	0.	0.000%	
39.404ms	0.	0.000%	
39.485ms	0.	0.000%	
39.567ms	0.	0.000%	
39.649ms	0.	0.000%	
39.731ms	0.	0.000%	
39.813ms	0.	0.000%	
39.895ms	0.	0.000%	
39.977ms	20.	32.258%	
40.059ms	6.	9.677%	
40.141ms	0.	0.000%	
40.223ms	0.	0.000%	
40.305ms	0.	0.000%	
	0.	0.000%	

(a) Traditional Interrupt Handling



up to	count	ratio	1% 2% 5% 10% 20% 50% 100
19.999ms	0.	0.000%	
19.999ms	0.	0.000%	
20.000ms	0.	0.000%	
20.001ms	0.	0.000%	
20.001ms	0.	0.000%	
20.002ms	0.	0.000%	
20.003ms	0.	0.000%	
20.003ms	0.	0.000%	
20.004ms	9.	11.538%	
20.004ms	50.	64.102%	
20.005ms	19.	24.358%	
20.006ms	0.	0.000%	
20.006ms	0.	0.000%	
20.007ms	0.	0.000%	
20.008ms	0.	0.000%	
20.008ms	0.	0.000%	
20.009ms	0.	0.000%	
	0.	0.000%	

(b) Parallel Interrupt Handling

图 5 任务 T1 的响应时间

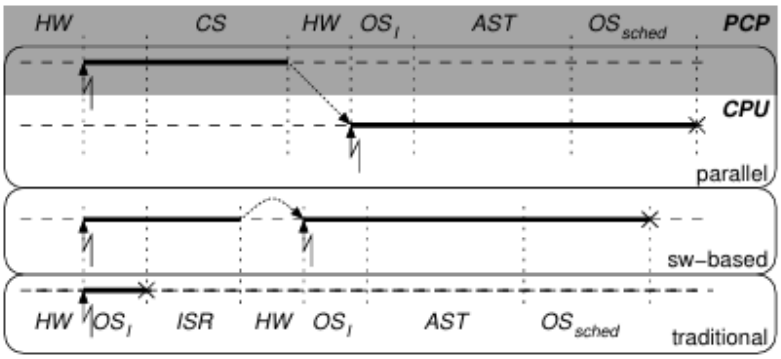
实验获取的，传统中断处理实现和我们原型的实现的响应时间分布分别如图 5（a）和（b）描述。可以明显地看出，传统实现的响应时间显著地更高，并且有明显的抖动。这些现象由 T1 被软实时的 ISR T2 干扰引起。如果我们以浮动的速率生成外部事件，抖动甚至会更大。但由于可行性的原因，我们以 2 ms 的固定周期触发 ISR，如表 1 所示。

与之相反，使用我们的原型实现的并行中断处理取得了明显更优的结果。响应时延低到几乎等于任务执行时间，以及很小的抖动（由测量误差引起）都表明 T1 不再被软实时任务干扰。我们同样能用之前提到的执行追踪证明这个干扰不再发生。

4.3 开销

表 2 事件处理函数的时延

	Best Case (bc)	**Worst Case** (wc)
Traditional	$0.900\mu s$	$6.660\mu s$
Software-Based	$9.062\mu s$	$14.822\mu s$
Parallel	$9.325\mu s$	$23.235\mu s$



HW	Hardware interrupt latency
OS_I	OS interrupt context management
OS_{sched}	OS scheduling and dispatching
ISR	Interrupt service routine
AST	Asynchronous system trap
CS	Critical sections CS_1 and CS_2

图 6 事件处理函数时延的组成

为了评估我们方法的代价，我们量化了我们方法相较于传统中断处理以及基于软件的方案的开销，使用的方式是比较在响应外部事件时的最好与最坏情况的时延。我们用一个等待信号量的线程来模拟基于软件的方案；该信号量由被外部事件触发的 ISR 发送。测量到的时延如表 2 所示。

这些时延的构成如图 6 所示。需要强调，硬件时延 HW 在每次出现时波动不大。然而，我们无法精确地确定这些硬件时延，因此硬件时延只是为了演示而提及。传统中断处理的最好情况时延只包括操作系统进行的中断上下文管理，也因此远远短于基于软件的方案和我们的并行中断处理。基于软件的方案和我们的并行中断处理实现只在最终触发 CPU 的 AST 的中断处理程序处不同。在基于软件的方案中，该中断处理程序在 CPU 上执行，而在我们的并行中断处理中，它在 PCP 上执行，导致了在最好情况下平均 263ns 的开销。

最坏情况下的时延，如表 2 所示，来源于事件处理函数可能经历的额外阻塞。出于公平比较的原因，我们假设所有实现都需要与 OS 内核同步，以保护 OS 内部的数据结构。传统中断处理和基于软件的方案的事件处理函数可能被最多一个向 OS 内核发出的系统调用阻塞。在基于软件的方案中，AST 自身不会被内核阻塞，因为它是在相同的处理器上运行的；因此，这一阻塞时间由时间最长的系统调用决定。在 CiAO OS 中，最长的系统调用耗时 *OSmax=5.760μs*，导致最坏情况的时延 *wc=bc+OSmax* 如图 2 所示。在并行中断处理中，PCP 中的中断处理函数和 AST 都可能需要等待调度器锁的获取。因此，事件处理函数可能被内核阻塞两次。因为 PCP 上的中断处理函数需要与其它覆盖执行的其它中断请求同步（见 2.1 章），在临界区 CS1 和 CS2 内，中断被锁定，导致了一个与临界区执行时间 *CS=2.390μs* 相等的额外阻塞时间。

如果我们分离地保证 CS1 和 CS2（同样见 2.1 章），PCP 上的中断处理函数可能在 CS2 的入口处被额外阻塞一次，使最坏情况时延 *wc=bc+3•OSmax+CS=28.995μs*。因此，连续地锁定 CS1 和 CS2 降低了最坏情况下的时延 5.760μs，而只是稍微增加了 CPU 的阻塞时间（小于 2.390μs）。

另一方面，在 CPU 上运行的线程也可能被在临界区 CS1 和 CS2 中拥有自旋锁的 PCP 阻塞。其可能经受的最大阻塞时间总计 $CS=2.390\mu s$，且在线程每次发出系统调用时都可能发生。这一阻塞带来的性能影响当然会随着 PCP 和 CPU 的性能差距增大而增大。为了应对这一点，我们保证 PCP 上执行的代码量尽可能小。此外，可以分离地保证临界区 CS1 和 CS2。在此方法中，我们用 CPU 的阻塞时间延长换取在响应外部事件时更优的最大时延。

因此，在最优情况下，我们的方案相较于基于软件的方案基本没有引入额外的开销，但它在一般的 OS 规范要求了更复杂的调度器功能的情况下，以及在栈共享的情况下，都更加灵活，正如 2.2 章所述。由于 PCP 和 CPU 间的同步需求增加，我们的并行中断处理在最坏情况下比基于软件的方案表现更差，但如果对可预测的系统行为和更复杂的功能有需求，这是一个可承受的代价。进一步地，需要注意，在并行中断处理中，最差情况发生的频率很低。它需要两个相邻发生的系统调用，而在基于软件的方案中，一个系统调用就会达到最差情况。

第 5 章 相关工作

中断为实时系统带来的可预测性问题是实时系统研究的一个特定领域的目标。许多方案试图忽略*多余的*中断请求[2, 18]。对于每个中断源，需要确定最小到达间隔时间、最大到达速率、最大连续传输长度等参数。之后，这些参数在一般的操作中被监测。不能适应这些给定参数边界条件的中断被视为*多余的*中断并被忽略。这样的方案对于防止和限制中断带来的过载情况很有效，但它们并未解决由二分裂的中断空间带来的问题。高优先级的硬实时任务仍然可能被干扰，例如被实现为 ISR 的软实时任务干扰。我们的方案在一定程度上也会受到 PCP 的中断过载的影响。然而，通过使用*以上*提到的技术，可以为 PCP 提供额外的，对中断过载的防护。在主处理器上执行的应用程序自身，可以自动防护过载的情况，正如那些统一优先级空间的方法。因为调度器控制了主处理器上的所有活动，可以采用零星服务器[20]等手段限制为异步事件的处理函数分配的时间。

不对称多处理器的概念当然在之前就已被操作系统社区使用。专门化的协处理器也已被用于在特定任务上支持操作系统。然而，这些方法的主要目标是最大化操作系统的吞吐率和效率。例如，流行的应用场景为消息传递系统中的延迟隐藏[3]或者网络协议卸载[11]。提供与我们相关的方案的操作系统内核是 Stankovic 和 Ramamritham 的 Spring 内核[21]。它在独立的 I/O 子系统中使用被称为前端处理器的不对称多处理器方法以卸载中断的处理。此方法可以有效避免单调速率优先级倒置。然而，Spring 内核的设计用途是只在非抢占的方式下运行任务。因此，它不适用于任何的实时系统——例如 AUTOSAR OS 的实现。

存在其它试图使用硬件抽象协助操作系统调度器的方法；然而，它们基本都依赖于*自定义硬件*。它们的方法——包括 cs2[15]、FASTCHART[13]、Silicon TRON[16]、HW-RTOS[4]和 Atalanta[22, 1]——通过在 FPGA 上综合特殊的电路，并且将其功能通过类似协处理器的方法提供，从而将操作系统的功能移动到硬件层面。它们中的一些方案，作为副产物，也防止了单调速率优先级倒置，但没有一个方案显式说明了这一点。与之不同，我们的方法没有在硬件中实现专用的调度器函数；相反，我们使用了*现有的商品硬件*实现统一的优先级空间（见 6.3 章），因此防止了单调速率优先级倒置。

第 6 章 讨论

在本章节中，我们讨论中断过载问题的可能后果以及它如何影响我们的方法、在 PCP 上直接执行线程和 ISR 的可能性，以及我们方法的通用性。

6.1 中断过载

正如在之前的章节中提到的，我们的方法依然会受到 PCP 上中断过载情况的影响——高优先级的中断会占满 PCP。然而，基于软件的方法，例如[5, 6]提出的，同样有这一问题。在那些方法中，需要使用一个小的 ISR 来通知与当前事件相关的任务——这个 ISR 可能会占满 CPU。

对两种方法都适用的一个可能的解决方案是将零星服务器算法扩展成也影响中断源。当服务器的执行预算耗尽，不仅需要调度器推迟对零星服务器之后的唤醒，还需要重新配置触发该零星服务器的中断源。这些中断源在没有执行预算时，需要被禁用或者降低优先级，取决于使用的零星服务器算法的具体实现。因此，该任务无法通过 ISR 消耗零星服务器分配的计算时间之外的计算时间，从而有效避免了中断过载。

6.2 在 PCP 上运行线程和 ISR

PCP 提供了一个与 CPU 完全独立的控制流，由此产生了将线程和 ISR 一同在 PCP 上运行的想法。然而，出于两个原因，我们没有这么做。

第一，在 PCP 上运行线程会在 PCP 中引入单调速率优先级倒置。考虑到我们使用 PCP 的目的是*防止*单调速率优先级倒置，这并不是合理的选择。即使将硬实时和软实时任务分离，将它们运行在不同的核心（例如，CPU 和 PCP）上，也不会完全消除单调速率优先级倒置。虽然软实时任务的中断是可以容忍的，但不同优先级的硬实时任务也会产生单调速率优先级倒置。因此，单独使用这个方法无法解决单调速率优先级倒置的问题，但它可以用于减少 CPU 和 PCP 的处理器间同步。

第二，PCP 的计算能力远小于 CPU，它是为很简单的任务，例如控制 DMA 传输或者中断预处理，而设计的，而我们使用 PCP 也出于这些目的。

6.3 通用性

我们已在 Infineon-TriCore 微控制器上的 CiAO 操作系统中实现了我们的方法。但该方法的概念本身可以用于十分广泛的硬件架构和 OS 内核。首先，几乎每个多核处理器或者多处理器系统都适合实现该概念，只要中断源的优先级可以在需要时配置。例如，汽车领域使用的很多微控制器的中断子系统都支持这些功能。其次，有很多其它的微控制器都配备了与 TriCore 架构的 PCP 相似的协处理器。例如 S12X 微控制器的 XGATE 协处理器[7]，或者 MPC5510 协处理器类和它们由 Freescale 提供的 I/O 协处理器[8]。

我们当前的实现，显然与实际使用的硬件绑定。但将我们的概念引入其它的架构，只需付出很少的努力。首先，它在操作系统中透明地实现，不会影响到应用程序本身。其次，只有很小一部分依赖于硬件的部分需要移植：在协处理器上运行的中断处理函数、通知 CPU 的机制，以及外设硬件初始化的组件。在我们的原型中，这些部分总共只有 187 行代码。这一数字主要源于 PCP 中断处理函数，因为它没有 C/C++编译器而使用汇编语言编写。

考虑到其对其它 OS 内核的适用性，任何事件驱动的 RTOS 内核都适合使用并行中断处理的概念从而维护统一的优先级空间。唯一的要求是需要一个特殊中断，用于触发 CPU 运行的内核的重新调度。

第 7 章 结论

我们证明了，可以利用商用嵌入式硬件，例如 Infineon-TriCore 平台，的协处理器阻止单调速率优先级倒置。相较于原有的基于软件的方案，我们的硬件支持的方法展现了重大的概念优势；这些优势包括进行栈共享的可行性、支持任务的多个激活实例，以及支持同一优先级的多种任务——例如，AUTOSAR 嵌入式 OS 标准要求这些内容。除此之外，通过评估我们方法在 CiAO 操作系统上的实现，我们证明了该概念杜绝了由低优先级事件的中断引发的干扰。我们的方案仅仅轻微增加了开销，而为了提高对任何实时系统都至关重要的可预测性，这只是一点小小的代价。

参考文献

[1] Bilge E. S. Akgul, Vincent J. Mooney III, Henrik Thane, and Pramote Kuacharoen. Hardware support for priority inheritance. In 24th IEEE Int. Symp. on Real-Time Systems (RTSS ’03), page 246, Washington, DC, USA, 2003. IEEE.

[2] AUTOSAR. Specification of operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

[3] Ulrich Brüning, Wolfgang K. Giloi, and Wolfgang Schröder-Preikschat. Latency hiding in message-passing architectures. In 8th Int. Symp. on Parallel Processing (IPPS ’94), pages 704–709, Washington, DC, USA, 1994. IEEE. 173

[4] Sathish Chandra, Francesco Regazzoni, and Marcello Lajolo. Hardware/software partitioning of operating systems: A behavioral synthesis approach. In Proceedings of the 16th ACMGreat Lakes Symposium on VLSI (GLSVLSI ’06), pages 324–329, New York, NY, USA, 2006. ACM.

[5] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In 12th IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS ’06), pages 14–23, Los Alamitos, CA, USA, 2006. IEEE.

[6] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt scheduling with low overhead for real-time kernels. In 12th IEEE Int. Conf. on Emb. and

RT Computing Systems and App. (RTCSA ’06), pages 385–394, Washington, DC, USA, 2006. IEEE.

[7] Freescale Semiconductor. XGATE Block Guide 02.09, November 2004.

[8] Freescale Semiconductor. MPC5510 Family Product Brief, September 2007.

[9] Micha Hofri. Proof of a mutual exclusion algorithm. SIGOPS Oper. Syst. Rev., 24(1):18–22, 1990.

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, 11th Eur. Conf. on OOP (ECOOP ’97), volume 1241 of LNCS, pages 220–242. Springer, June 1997.

[11] Hyong-Youb Kim and Scott Rixner. Connection handoff policies for TCP offload network interfaces. In 7th Symp. on OS Design and Implementation (OSDI ’06), pages 293–306,

Berkeley, CA, USA, 2006. USENIX.

[12] David Lake. Stack usage in computer-related operating systems. US Patent No. 722543, May 2007.

[13] Lennart Lindh and Frank Stanischewski. FASTCHART– a fast time deterministic CPU and hardware based real-time-kernel. In Proceedings of the 1991 Euromicro Workshop on Real-Time Systems, pages 36–40, Jun 1991.

[14] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In 2009 USENIX TC, pages 215–228, Berkeley, CA, USA, June 2009. USENIX.

[15] Andrew Morton and Wayne M. Loucks. A hardware/software kernel for system on chip designs. In Proceedings of the 2004 ACM Symposium on Applied Computing (SAC ’04), pages 869–875, New York, NY, USA, 2004. ACM.

[16] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. Hardware implementation of a real-time operating system. In Proceedings of the 12th TRON Project International Symposium (TRON '95), pages 34–42, Nov 1995.

[17] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. http://portal.osek-vdx.org/files/pdf/ specs/os223.pdf, visited 2009-03-26.

[18] John Regehr and Usit Duongsaa. Preventing interrupt overload. In 2005 Languages, Compilers and Tools for Embedded Systems (LCTES '05), pages 50–58, New York, NY, USA, 2005. ACM.

[19] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software, 20(7):636–651, 2007.

[20] Brinkley Sprunt, Lui Sha, and John P. Lehoczky. Aperiodic task scheduling for hard real-time systems. Real-Time Systems Journal, 1(1):27–60, 1989.

[21] John A. Stankovic and Krithi Ramamritham. The Spring kernel: A new paradigm for real-time operating systems. ACMOSR,27(3):54–71, July 1989.

[22] Di-Shi Sun, Douglas M. Blough, and Vincent John Mooney III. Atalanta: A new multiprocessor RTOS kernel for system-on-a-chip applications. Technical report, Georgia Institute of Technology, 2002