LEFT-RS：面向多核实时系统的无锁容错资源共享协议

Nan Chen*、Xiaotian Dai*、Tong Cheng†、Alan Burns*、Iain Bate*、Shuai Zhao†
*约克大学，英国 †中山大学，中国

arXiv:2512.21701v1 [cs.OS] 25 Dec 2025

摘要—**新兴实时应用推动了向多核嵌入式系统的转型。在这些系统中，由于功能需求和可用性有限，任务必须共享资源。这些资源无论是局部资源还是全局资源，都在临界区内受到保护，以防止竞争条件；锁协议则确保互斥访问和时序要求。然而，发生在临界区内的瞬态故障可能会中断执行，并在多个任务之间传播错误。传统锁协议无法应对这类故障，而整合传统容错技术通常会增加阻塞。近期方法通过并行副本执行改善了故障恢复能力，但由于顺序访问、协调开销以及易受共模故障影响，仍然面临挑战。本文提出了一种面向多核实时系统的无锁容错资源共享（LEFT-RS）协议。LEFT-RS允许任务并发访问和读取全局资源，同时并行进入各自的临界区。如果其他任务发生故障，某个任务在执行成功后即可提前完成访问，从而提高资源使用效率。我们的设计还限制了开销并增强了故障恢复能力。我们提出了全面的最坏情况响应时间分析，以确保时序保证。广泛的评估结果表明，我们的方法显著优于现有方法，平均可将可调度性提高最多**84．5%。

I. 引言

随着新兴实时应用需求不断增长，系统正从单核向多核嵌入式系统转变。在这类系统中，由于功能需求和固有的资源约束，任务经常需要访问共享资源。这些共享资源包括本地资源和全局资源。本地共享资源位于单个核心内，但该核心上的多个线程都可能访问它们，例如每核心队列或同步标志；而全局共享资源则跨越多个核心，包括内存映射 I/O 设备和系统级数据结构。为避免竞争条件，此类资源通常通过临界区加以保护。临界区是需要互斥的代码片段。资源锁协议旨在确保任务互斥访问资源，同时满足系统的时序要求 [1]。

关键领域中的嵌入式系统必须在存在故障的情况下继续正确运行。瞬态故障是指在执行过程中因硬件波动、时序异常或其他短暂状况而发生的临时错误，尤其值得关注 [2]。通常可通过时间冗余（例如重新执行或检查点技术，其中检查点技术支持小范围重新执行），或空间冗余（例如复制）来容忍此类故障 [3]。必须处理临界区期间发生的瞬态故障，以防错误在任务间传播。传统的资源锁协议并未明确考虑容错 [1]。将检查点技术直接集成到资源共享协议中，可能会显著加剧全局资源竞争 [4]，这主要是因为故障恢复可能大幅延长临界区的执行时间，导致锁被持有更长时间，从而增加系统中其他任务的阻塞时间。

为应对这一挑战，MSRP-FT [4] 结合了检查点和复制，使自旋任务能够并行执行副本，从而减少临界区执行期间因瞬态故障而损失的时间。然而，资源请求仍按照先进先出顺序依次处理，在故障频发时可能导致长时间阻塞。此外，协调副本执行会带来额外开销，而依赖相同副本则使系统容易受到相关故障或时序敏感故障的影响。这些局限性将在第 III-C 节中详细讨论。

本文提出了一种面向多核实时系统的无锁容错资源共享协议，称为 LEFT-RS。我们使用先进先出队列记录资源请求顺序。与 MSRP-FT 中按顺序管理资源访问的方式不同，LEFT-RS 采用无锁设计，使任务能够并发读取全局资源并启动本地执行。当任务发生故障或共享资源更新时，任务会重新执行其临界区，以确保无故障执行并避免竞争条件。如果先进先出队列中的前序任务发生故障，每个任务在成功执行后即可完成访问。LEFT-RS 提高了资源访问效率，并确保随着故障发生次数增加，远程阻塞（由远程核心上的任务因资源竞争而导致的阻塞）不会加剧。此外，其设计避免了高昂的协调开销，并增强了故障恢复能力。为保证时序可预测性，我们提供了全面的最坏情况响应时间分析。大量实验评估表明，在可调度性方面，我们的方法平均优于最先进的方法，最高可达 84.5%。

后续各节的结构如下：第II节介绍系统模型。第III 节回顾相关工作，并深入分析当前最先进方法的局限性。第IV 节介绍所提出的协议，而第V 节则与

通讯作者：Shuai Zhao，zhaosh56@mail.sysu.edu.cn

当前最先进的方法进行理论比较。第VI 节给出最坏情况下的响应时间分析。第VII节展示评估结果，第VIII 节总结全文。

II. 系统模型

我们考虑由一组相同核心（记为 Λ）和一组偶发任务（记为 Γ）组成的系统，并采用 FP-FPS（完全分区固定优先级调度）方案进行调度。 Λ 中的每个核心表示为 λ_k，其中 k标识核心的索引。每个任务 τ_i （Γ中的第 i^{th} 个任务）由元组 $\tau_i = \{C_i, T_i, D_i, P_i\}$刻画。其中，$C_i$ 表示不访问共享资源时的纯最坏执行时间（WCET）， T_i 表示周期（或最小到达间隔）， D_i 表示满足 $D_i \leq T_i$的受限截止时间， P_i 表示任务的优先级。每个任务都被分配一个唯一的优先级，P_i 的值越大表示优先级越高。对于给定任务 τ_i，我们分别将同一核心上优先级高于和低于该任务的任务记为 τ_h 和 τ_l。分配到其他核心的任务记为 τ_j。

系统还包含一组共享资源，记为 $\mathcal{R}$，其中 x^{th} 共享资源表示为 r^x。每个资源 r^x 由两个参数 c^x 和 N_i^x描述。参数 c^x 表示与 r^x相关的计算长度，而 N_i^x 表示任务 τ_i 在单次释放期间向 r^x 发出的请求次数。本文不考虑嵌套资源共享，即任务一次只能持有一个资源。不过，系统可以直接支持组锁 [5] 。

本文研究瞬态故障，这类故障可以通过检查点或复制等冗余技术加以缓解。假设每次瞬态故障只影响一个任务。故障检测在指定的检查点处进行，并通过验收测试（例如结果验证或一致性检查）来验证任务输出的正确性 [6]。瞬态故障的发生频率可以通过多种方式建模 [7],，但这不属于本文的讨论范围。与 [4], 类似，我们假设每个任务在单次释放期间最多可能遭遇 f_i 次故障。任务在访问 r^x 期间可能遭遇的故障次数记为 $f_i^x \leq f_i$。我们用 n_i^x 表示 τ_i 向 r^x发出的请求的总执行次数，其中包括 f_i^x次失败执行和一次成功执行（$n_i^x = f_i^x + 1$）。所有符号汇总于表 I。

表 I：系统模型中使用的符号列表

符号	说明
Λ	系统中相同核心的集合
λ_k	索引为 k, $\lambda_k \in \Lambda$ 的核心
Γ	偶发任务集合
τ_i	索引为 i, $\tau_i \in \Gamma$ 的任务
τ_h, τ_l, τ_j	与 τ_i相关的任务：在其上具有高/低优先级同一核心，或不同核心
C_i	τ_i的纯最坏执行时间（WCET）
T_i, D_i	周期（最小到达间隔）和受限截止期限（$D_i \leq T_i$）的 τ_i
P_i	τ_i 的优先级（ P_i 越大，优先级越高）
$\mathcal{R}$	系统中的共享资源集合
r^x	索引为 x 的共享资源
c^x	r^x 的计算长度
N_i^x	每次释放时 τ_i 对 r^x 的请求数量
f_i, f_i^x, n_i^x	τ_i 每次释放时发生的故障数量。故障以及访问 r^x期间的总执行次数

III. 相关工作与背景

本节首先介绍资源共享协议方面的文献，然后回顾具备故障感知能力的资源共享领域的最新进展。

A. 资源共享协议

基于锁的协议已在多核系统中得到广泛研究，可实现高效的资源共享。多处理器优先级上限协议（MPCP）将传统的优先级上限扩展，以减少远程阻塞 [8]。多处理器栈资源协议（MSRP）采用非抢占式自旋来避免抢占延迟 [8]。相比之下，多处理器资源共享协议（MrsP）采用全局优先级上限来限定阻塞，引入协助机制，并允许跨核迁移 [9]。灵活多处理器锁定协议（FMLP）区分短临界区和长临界区，结合自旋与挂起来平衡阻塞和并发 [10]。$O(m)$锁定协议（OMLP）进一步改进基于优先级的机制，以提升可扩展性 [11], ；灵活资源访问协议（FRAP）则引入动态自旋优先级和先进的阻塞分析，以提升性能 [12]。每种协议都针对特定挑战，其有效性取决于系统架构和应用需求。

相比之下，无锁方法 [13]依赖原子操作（如比较并交换（CAS）或加载链接/条件存储（LL/SC）），在不使用显式锁的情况下更新共享资源。这些算法不会通过串行化访问来强制实现互斥，而是通过监控共享数据的变化来检测冲突，并在必要时重试操作。这种非阻塞设计可确保整个系统持续取得进展，同时防止竞争条件和数据损坏。

B. 故障感知资源共享

[14] 中的工作解决了临界区中的瞬态故障问题，但采用简单的检查点机制，通过反复顺序重新执行资源请求来容忍故障。在传统MSRP的基础上， [15], MSRP- FT [4] 能够高效容忍混合关键性系统（MCS）中任务临界区内的瞬态故障。由于MSRP-FT的容错方法独立于MCS，我们将在标准系统配置下讨论该方法，以便于阐述。

如图 1所示，MSRP-FT 不仅在每个任务的开始和结束处设置检查点，还会在任务执行期间的临界区周围设置检查点。这些检查点将执行划分为普通区段（不涉及共享资源）和临界区（访问共享资源）。临界区包括本地（核内）和全局（核间）资源访问。每个检查点都会检测前一段中的故障，并为下一段保存状态（如适用）。这种设置使系统能够在临界区执行完毕后及时容忍故障，防止传递性错误影响其他任务，并避免不必要的大规模重新执行。

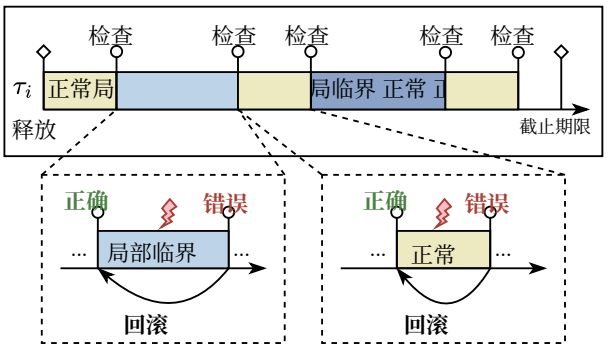


图 1：普通区段和局部临界区的容错

上一段中的故障，并为下一段保存状态（如适用）。这种设置使系统能够在临界区执行完毕后及时容忍故障，防止传递性错误影响其他任务，并避免不必要的大规模重新执行。

如果在普通区段中检测到故障，系统会回滚到最近的检查点，并仅重新执行发生故障的片段。管理本地共享资源时，采用与 MSRP 相同的上限协议。每个本地资源 r^x 的上限优先级设为访问该资源的所有任务中的最高优先级。当任务锁定本地资源时，其优先级会暂时提升至该资源的上限优先级。如图 1所示，所采用的容错方法与普通区段相同，即系统启动回滚和重新执行。

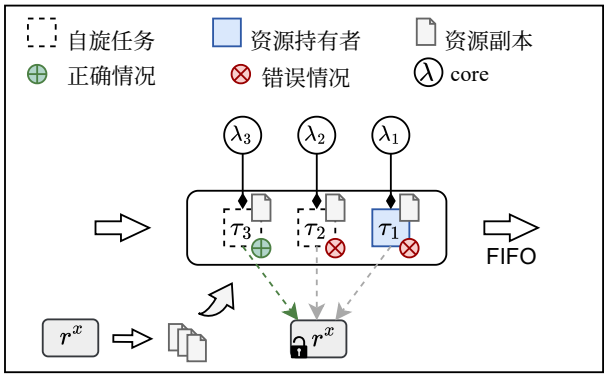


图 2：全局临界区的容错

此外，MSRP-FT 还引入了一种专为全局资源设计的新型容错机制。如图 2所示，当任务请求全局资源时，会进入一个先进先出队列，资源按先进先出顺序分配。队列头部的任务（即 τ_1）成为资源持有者，而其他任务则保持在忙等待状态并进行自旋。资源持有者读取共享资源，并将其执行复制到多个副本中，使所有自旋任务（即 τ_2 和 τ_3）都能协助并行执行其临界区。副本在本地执行；检测到故障后，会在同一核心上重新执行该副本。资源更新作为原子操作执行，只有无故障执行（即 τ_3）才有资格用于更新

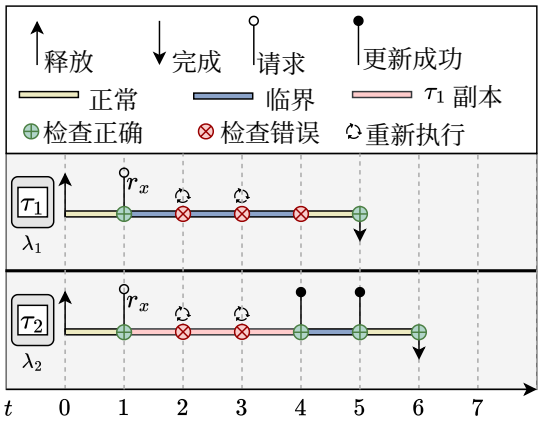


图 3：MSPR-FT 下 τ_2 的最坏情况资源访问

资源。资源更新后，所有副本都会终止，队列头部的任务退出队列，接下来的任务（即 τ_2）重复这一过程。

任务执行其所有n_j^x 执行所需的时间计算为 $\lceil n_j^x/k_j \rceil \cdot c^x$ [4]，其中 k_j表示可用执行单元的数量（包括该任务本身以及在先进先出队列中排在其后的辅助任务）。为了直观地展示 MSRP-FT 的性能，我们给出以下示例，以说明 τ_2的最坏情况资源访问时间。

示例 *1*。如图 *3* 所示，τ_1 以 $n_1^x = 6$ 在 λ_1上执行，而 τ_2 以 $n_2^x = 1$ 在 λ_2上执行。在时刻 $t = 1$，两个任务同时请求访问 r^x。临界区长度为 $c^x = 1$。τ_1 成为先进先出队列的队首，τ_2 随后排队。两者开始并发执行各自的临界区，其中 τ_2 通过执行 τ_1的临界区副本提供辅助。经过多次失败和重试后，τ_2在 $t = 4$成功完成第 *6* 次执行，更新资源，并帮助 τ_1 完成，使得 τ_1 在 $t = 4$ 离开先进先出队列，最终于 $t = 5$完成。在 $t = 4$，τ_2 随后执行自己的临界区。由于队列中没有其他任务，它独自执行，并在 $n_2^x = 1$后成功完成，于 $t = 5$更新资源。τ_2 最终于 $t = 6$完成。

在 MSRP-FT 下，不考虑协调开销时，给定任务的最坏情况资源访问时间τ_i （例如，τ_2 示例1）由以下两个条件决定。首先，τ_i 会遭受最大可能的远程阻塞（因全局资源竞争而受到远程核心的阻塞），因为它被放置在先进先出队列的末尾；此时没有可用的外部辅助核心，这意味着它必须在自己的核心上依次执行其 n_i^x 副本。其次，先进先出队列中排在 m前面的远程请求的最大数量（记为 τ_i ）按如下方式排列：执行次数较多的任务 n_j^x安排得更靠近队列末尾，且可用的辅助者更少。具体而言，第一个远程请求（其n_j^x最大）仅由 $k_1 = 2$ 个核心（包括其自身和 τ_i）提供辅助，第二个由 $k_2 = 3$个核心提供辅助，依此类推。请求 τ_i 的最坏情况访问时间如公式（1） [4]所示。所有项均以 c^x为单位，即单个临界区的执行长度

资源 r^x 的单个临界区的执行长度。具体而言， n_i^x 是由故障导致的重新执行次数， S_i^x 是由远程请求造成的阻塞延迟。我们进一步观察到，每个远程请求至少会使阻塞增加一个单位的 c^x ，因为 $\lceil n_j^x/k_j \rceil \geq 1$。因此，在最坏情况下， $S_i^x \geq m$。

$$E_i^x = (n_i^x + S_i^x) \cdot c^x, \quad S_i^x \geq m \tag{1}$$

基于 MSRP-FT 的特性，我们提出以下推论，后文将会用到。

推论1。 在 *MSRP-FT* 下，给定任务 τ_i 遭受的远程阻塞单元 S_i^x会随着远程请求的 n_j^x 增加而累积。更具体地说，如果多个远程请求具有 $\lceil n_j^x/k_j \rceil > 1$，那么 $S_i^x > m+1$。例如，如果一个具有 $k_j = 2$ 的任务随后遭遇 *2* 次故障（即$n_j^x = 3$），则满足 $\lceil 3/2 \rceil = 2 > 1$，从而产生多个远程阻塞单元。

C. 现有技术的局限性

MSRP-FT 协议利用副本的并行执行提高了容错效率。尽管这种方法提高了故障恢复效率，但也带来了以下新挑战。

局限1。 当故障数量较多时，即使采用帮助机制，依然按顺序管理资源访问也可能导致显著延迟。

尽管 MSRP-FT 通过允许自旋任务并行执行副本来加速容错，但每个任务仍必须按顺序等待资源访问。根据推论1，当远程任务频繁发生故障（即 n_j^x 较高）时，它可能会显著延迟先进先出队列中排在其后的任务的执行，即增加远程阻塞（$S_i^x \cdot c^x$）。

局限2。 利用自旋任务来提供辅助需要复杂的调度，并可能产生大量开销。

MSRP-FT 由于其辅助机制而引入了不可避免的协调开销。当任务位于先进先出队列的队首时，必须构造一个共享操作描述符，我们称之为封装，其中包括用于临界区逻辑的函数指针、对共享资源的引用，以及任何相关的执行上下文 [16], [17]。该结构会被写入全局可见的内存位置，例如动态随机存取存储器或末级缓存，并使用内存屏障设置就绪标志，以确保跨核可见性 [18]。此步骤会产生每个队首任务的元数据设置成本，记为 O_{wrap}。

队列中位于队首之后的每个任务都必须监测共享就绪标志，并在资源可用后获取封装，在本地复制共享资源，然后执行队首任务的临界区。这种协作执行会引入每个副本的开销 O_{replica}，其中包括协调、本地准备和控制转移成本 [16], [17]。当该任务随后自身成为队首时，还必须构建自己的封装，从而产生一次性设置成本，记为 $O_{\text{self_wrap}}$。因此，如果一个任务在先进先出队列中前面有 {v4}个其他任务，则它为单个资源请求产生的总开销为：

$$O_{\text{total}}(m) = m \cdot (O_{\text{wrap}} + O_{\text{replica}}) + O_{\text{self_wrap}} \tag{2}$$

局限3。 基于副本的并行执行依赖于相同的执行逻辑以及同一时刻同步的数据，因此可能无法有效应对确定性故障或共模故障。

在 MSRP-FT 中，所有副本都执行源自同一包装描述符的相同操作，从而导致高度同步的控制流和内存访问模式。因此，由特定执行顺序或时序干扰触发的故障可能会同时影响所有副本，导致协同失效。

IV. LEFT-RS 协议

本节针对多核实时系统提出 LEFT-RS。LEFT-RS 引入了新机制，以更高效地处理访问全局资源时发生的故障，旨在提高系统的可调度性。我们首先介绍基本设置，其中包括不涉及全局资源共享的部分。然后，我们说明 LEFT-RS 如何管理存在故障时的全局资源共享，并阐述其设计依据。

A. 普通临界区和本地临界区

为避免传递性错误，并防止因重新执行整个任务而导致不必要的资源竞争，我们继续采用 MSRP-FT 中的检查点机制，如第 III-B 节所示。检查点不仅设置在每个任务的开始和结束处，还设置在任务执行过程中的临界区周围。这些检查点将执行划分为普通区段（不涉及共享资源）和临界区段（全局临界区和局部临界区）。每个检查点在适用时都会检测前一段中的故障，并保存下一段所需的状态。当任务包含很少或不包含临界区，从而形成较大的普通区段时，可以引入检查点对其进行细分，以缩小重新执行范围。然而，由于我们关注的是临界区中的容错，因此不再进一步探讨普通区段内的检查点设置。

如果在正常或局部临界区中发生故障，系统只需执行回滚并重新执行。对于本地资源的管理，我们假设采用优先级上限协议：访问资源的任务会将其优先级提升至上限优先级，该优先级等于访问同一资源的本地任务中的最高优先级。

B. 全局资源共享

在本小节中，我们将通过一系列规则介绍 LEFT-RS 如何管理全局资源共享。这些规则的介绍顺序并不遵循任务访问共享资源时的执行顺序，而是依据设计思路。最后，示例2 将展示任务遵循这些规则时的完整执行顺序概览。

该协议的核心目标是在容错条件下提高资源访问效率。为此，该协议旨在让未发生故障而顺利完成的任务，在其他任务遇到故障时能够更早更新全局资源。这样可以减少不必要的延迟，提高整体系统吞吐量。实现临界区并发执行是达成这一目标的关键。正如规则 1所述，这通过无锁方法实现；由于并发读取不会修改共享状态，因此该方法是可行的。

规则1。 请求全局资源时，允许所有任务同时读取该资源，并在本地进入各自的临界区，而无需获取全局锁。

虽然并发执行提高了效率，但也带来了两个必须解决的关键挑战：**(1)** 对全局资源的并发更新可能导致数据竞争；**(2)** 使用陈旧数据，即由于更新可见性有限，任务基于过时信息运行。

第一个并发问题通过规则 2和3得到缓解。在规则2中，先进先出队列记录任务请求的顺序，以规范更新顺序，而不是像 MSRP-FT 那样强制规定资源访问顺序。规则 3 确保按照先进先出顺序授予更新优先级，以保证公平性，并将资源更新限制为一次仅由一个任务执行，从而避免竞争条件。

规则2。 任务访问全局资源时，会将其加入先进先出队列。

规则3。 成功（无故障）执行后，如果多个任务并发请求更新，则按照先进先出顺序授予更新，并且更新必须以原子方式执行。

第二个并发问题由规则 4解决，该规则确保每次更新都会触发先进先出队列中其他任务重新读取并重新执行，从而避免过时执行。

规则 4。 任务成功更新全局资源后，便会离开先进先出队列。所有基于资源过时副本运行的其他任务都会收到通知，中止当前执行、丢弃本地副本，并使用更新后的全局资源重新开始其临界区。

此外，根据规则5的定义，任务也会因在临界区期间遇到瞬态故障而重新执行。这会导致临界区出现两种重新执行行为，如定义1和2所示。

规则 5。 执行失败（发生故障）时，任务会单独重新执行发生故障的临界区。

定义1。 *[数据诱发的重新执行]* 当另一个任务更新共享资源，使当前任务的本地副本失效时，就会触发重新执行；当前任务需要使用最新数据重新开始。

定义2。 *[故障诱发的重新执行]* 当任务在临界区期间检测到瞬态故障时，就会触发重新执行；任务需要重试以确保正确性。

主要的并发问题通过规范并发更新和数据一致性维护得以解决。然而，由于请求到达和执行路径的差异，资源访问可能在不同时间开始和完成，任务可能会以乱序方式完成执行。因此，规范更新和重新执行的时机对于防止不公平和不可预测的延迟仍然至关重要。为解决这一问题，该协议引入了一套全面的同步机制，由三条紧密集成的规则组成。首先，规则 6 确保在不同时间加入先进先出的任务同时开始执行，使其执行窗口保持一致，并减少初始时间差。

规则 6. 加入先进先出队列后，如果队列头部的任务正处于临界区执行过程中，后续任务将进入同步期，并等待头部任务完成当前执行；否则将跳过同步步骤。

然而，任务仍可能比预期更早完成临界区。例如，考虑以下场景：τ_a、τ_b和 τ_c 按从队首到队尾的顺序排列在先进先出队列中。如果 τ_c 提前完成并更新资源，可能会触发由数据引起的 τ_a 和 τ_b重新执行。此外，后来加入的任务也可能因此导致 τ_a 和 τ_b 饥饿。为解决这一问题，规则7 规定任务成功执行后对共享资源的更新方式。即使任务提前完成，在所有排在它前面的任务均已完成或发生故障之前，也不能更新全局资源。这样可以防止提前完成触发对前序任务不必要的、由数据引起的重新执行，并确保更新公平性，从而补全规则3。

规则 7。 成功（无故障）执行后，先进先出队列中的队首任务可以直接尝试更新共享资源。其他任务（队首任务除外）只有在确认所有排在自己前面的任务在当前执行过程中都遇到故障后，才能尝试更新。

在此情形下，假设 τ_c 成功完成。根据规则 7，它必须等待 τ_a 和 τ_b 完成后才能更新。如果 τ_a 先完成，但遇到故障并立即重新执行，那么当 τ_b 最终完成时，τ_c 仍必须等待 τ_a的新一轮执行，这会带来额外延迟。因此，规则 8 通过控制故障诱发的重新执行时机，对规则 5进行补充。这可以防止重新执行的任务干扰后续任务的更新，避免因未同步的重试造成串行延迟。

规则 8。 执行失败（出现故障）后，该任务会等待先进先出队列中的所有任务完成当前执行，然后独自重新执行发生故障的临界区。

这些设计元素共同实现了两个核心原则，用以支配整个协议的行为。

- 理由 *1*：由于前序任务更新了先进先出队列中的全局资源，因此任务需要重新执行。
- 理由 *2*：任务会因后到任务所做的更新而重新执行。然而，这类重新执行与故障诱发的重新执行重叠。

理由1可确保公平性和重新执行次数有界：任务因其在先进先出队列中的相对位置而重新执行。根据规则 4，对全局资源的每次更新都会触发一个任务离开先进先出队列。因此，重新执行的次数是可预测的。理由 2 对于在容错条件下实现高效的资源访问至关重要。如果前序任务遇到故障，它无论如何都会重新执行。如果此次重新执行与后到任务所做的资源更新重叠，就能通过减少延迟使后到任务受益，同时不会对前序任务造成不利影响。由于只有当任务比预期更早完成时才会出现强制等待时间，因此执行时间与等待时间之和仍在单个临界区的持续时间范围内，这一点将在引理 4中进一步证明。

最后，规则9 和10 完善了该协议。规则9 可通过将活动优先级提升至系统最高级别来实现，从而确保任务在访问全局资源期间不会被抢占。这一行为继承自 MSRP [15]，选择它是因为其简单性。虽然基于优先级上限的方法（例如 MrsP） [9]，可能会进一步减少到达阻塞，但这超出了本研究的范围。

规则 9。当任务请求全局资源时，它相对于本地任务变为非抢占式，并持续保持该状态，直到完成资源访问。

规则10 通过消除不必要的延迟完善了规则6 。我们提出引理1来支持规则10。

规则 10。如果任务在加入先进先出队列时能够确保所有排在它前面的任务都不会发生故障（即 $n_j^x = 1$，或者这些任务已经遭遇了允许的最大故障次数），或者系统的容错模式已禁用，则可以跳过初始同步期。

引理 1. 对于给定的任务 τ_i，如果先进先出队列中排在它前面的任务不会发生故障，那么 τ_i 的请求无需进入同步期。

证明。 由于排在前面的任务必须不晚于 τ_i 开始，因此必须在执行周期 c^x 内为排在前面的任务进行资源更新，因为这些任务不会遇到故障。即使 τ_i 提前完成，规则7 也足以保证任务只能在排在前面的任务完成后更新资源，这与理由 1 和理由 2 一致。 □

这些规则通过示例2 进行可视化，该示例展示了一个全局资源请求的完整执行序列
LEFT-RS 下的资源。

示例 2. 如图 4所示，τ_1 和 τ_2 分别在核心 λ_1 和 λ_2上执行。在 $t = 1$，τ_1 请求访问 r^x，其 $c^x = 2$。它随即变为本地非抢占式，并立即开始执行其临界区（规则 1和 9），无需经过同步期（规则 6）。在 $t = 2$，τ_2 请求访问 r^x，此时先进先出队列的顺序变为 τ_1 和 τ_2，从队首到队尾。根据规则 6，由于 τ_1 此时正处于临界区执行过程的中间

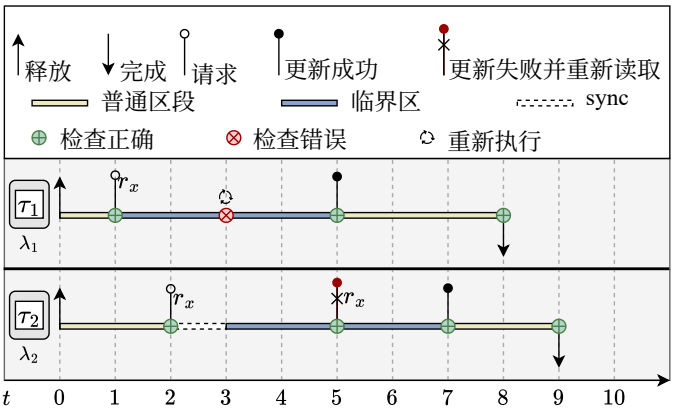


图 4：LEFT-RS 全局资源管理示例

$t = 2$，τ_2 进入同步期。在 $t = 3$，由于 τ_1执行失败且所有执行均已同步，τ_1 重新执行（规则 *5* 和 *8*），而 τ_2 读取共享资源，并开始与 τ_1并行执行其临界区（规则 *1*）。在 $t = 5$，τ_1 和 τ_2 的执行均正确且已同步，τ_1 按照更新过程成功更新 r^x（规则 *3* 和 *7*）。在 $t = 5$，资源更新后，τ_1 离开先进先出队列（规则 *4*），并于 $t = 8$完成执行。此外，在 $t = 5$，系统向 τ_2 发出信号，要求其重新读取并重新执行共享资源（规则 *4*）。在$t = 7$，τ_2 成功更新 r^x（规则 *3* 和 *7*），离开先进先出队列（规则 *4*），并于 $t = 9$完成。

总的来说，LEFT-RS 允许任务并发开始资源访问，而不必按先进先出顺序获得服务。如果前序任务发生故障，任务成功执行后即可提前退出先进先出队列。这种设计能够有效解决局限1。此外，LEFT-RS 通过消除协作式副本执行，避免了方程（2）中所述的协调开销。任务既不构建或发布共享包装描述符，也不协助执行其他任务的临界区。相反，每个任务在获取共享资源状态后，独立执行自己的临界区，从而解决局限2。由于本地故障检测和线程通信等其他开销对两种协议而言都是共有的，或对性能影响微乎其微，因此不在此讨论。由于 LEFT-RS 下的任务执行各自的逻辑和执行路径，影响某个任务的故障不太可能传播至其他任务。这种设计从根本上降低了同步故障的风险，并增强了并发执行中的故障隔离，从而解决局限3。第 V 节对资源访问进行的最坏情况分析将进一步展示所提规则及其原理的有效性。

C. 实现讨论

无锁方法已得到充分确立，并有着悠久的历史 [13]。LEFT-RS 适用于这样的共享资源：任务可以读取一致的快照，并在进行一次原子更新之前执行本地计算。典型示例包括

状态标志、控制寄存器、配置参数和共享计数器。这些结构在嵌入式系统中十分常见，也符合 LEFT-RS 的设计理念：支持安全的并发读取，并避免复杂的协调。此外，LEFT-RS 和 MSRP-FT [4] 都要求进行原子更新，并在本地并发执行临界区，因此它们在理论上的适用范围相同。

LEFT-RS 要求每个任务将共享资源复制到其私有内存（例如 L1/L2 缓存）中并独立执行；大多数商用现成（COTS）架构都支持这一点。从软件角度来看，规则 4 涉及向其他任务发送信号通知，而规则6、7和8 则涉及检查其他任务的执行状态。这种协调可以通过轻量级通信机制实现，例如共享状态标志或条件信号通知。

V. 与 MSRP-FT 的理论比较

所提出的方法与 MSRP-FT 的关键区别在于全局资源访问的管理。在进行完整分析之前，本节通过考察给定任务 τ_i 对全局资源 r^x的一次请求的最坏情况资源访问时间，分析所提出方法的优势。

根据规则 5，在所提出的方法下，每个任务遇到故障时都会在没有协助的情况下独立重新执行。因此，在最坏情况下，若不考虑远程任务造成的延迟，来自 τ_i 的资源请求将执行 n_i^x 次其临界区，其中包括 f_i^x次执行失败和一次成功提交。

根据规则 4 和 7，每个任务都会因先进先出队列中前序任务的资源更新而执行数据诱发的重新执行。每次资源更新都会导致一个任务离开先进先出队列。因此，数据诱发的重新执行次数可由以下引理限定。

引理 2。 对于从任务 τ_i 到 r^x的给定请求，如果最多有 m $(\leq |\Lambda|)$ 个远程请求可以同时请求 r^x，则 τ_i 的请求最多可能引发 m 次数据诱发的重新执行。

证明。 数据诱发的重新执行是由于系统中的一次成功资源更新而发生的。同时，每次更新都会从先进先出队列中移除一个任务（规则 4）。队列中最多有 m 个前序任务实例，因此， τ_i 的请求最多可能因前序任务而引发 m 次数据诱发的重新执行。根据规则7，较晚到达的任务只有在引发故障时，才会对 τ_i造成数据诱发的重新执行。这已计入因故障导致的 f_i^x 次执行失败中。 □

根据规则 6，每个任务在请求资源时都可能进入同步时段。我们提出以下引理来界定同步时段。

引理 3。 给定资源请求的最大同步开销为一个 c^x 单位。

证明。资源请求加入先进先出队列并等待队首任务完成其当前执行时，会产生同步开销。由于最大执行长度为 c^x ，且同步最多发生一次，因此最大同步开销为 c^x。 □

引理 4。 同步开销是唯一额外的等待时间。

证明。 根据规则 7，任务在正确执行时可能需要等待前置任务完成。然而，规则 6强制所有任务同时开始；因此，等待时段仅在任务提前完成时发生，其执行时间与等待时间之和不会超过 c^x。规则 8同样适用：当任务发生故障时，它会等待所有任务完成后再执行重新执行。 □

根据引理 2、3 和4，我们可以得到以下推论。在推论 2 中， n_i^x 是不考虑远程核心延迟时 $\tau_i's$ 请求的基本最坏情况执行时间。 m考虑了数据导致的重新执行，如引理 2所示。 c^x的 1 个单位是同步开销，如引理3所示。

推论 2. 对于 *LEFT-RS* 下从 r^x 到 τ_i 的一项给定资源请求，其最坏情况资源访问时间的上界为 $E_i^x = (n_i^x + m + 1) \cdot c^x$。

根据引理1，当前面任务的 τ_i 在先进先出队列中未遭遇故障时，同步开销可能不会发生，因此可得以下推论。

推论 3. 对于 τ_i 到 r^x的一项给定资源请求，如果先进先出队列中的前序任务未发生故障，则采用所提出方法时的最坏情况资源访问时间为 $E_i^x = (n_i^x + m) \cdot c^x$。

如公式（1）所示，MSRP-FT 下来自 τ_i 的一项给定请求的最坏情况资源访问时间为 $E_i^x \geq (n_i^x + m) \cdot c^x$，这表明在有限的场景下，该时间可能优于（即小于）所提出的上界 $E_i^x = (n_i^x + m + 1) \cdot c^x$ 。更具体地说，根据推论1和2，可得以下推论。

推论 4. 对于从 τ_i 到 r^x的请求，随着故障数量增加，*LEFT-RS* 的性能优于 *MSRP-FT*。更具体地说，当满足 $\lceil n_j^x / k_j \rceil > 1$ 的远程请求超过一个时，这一结论成立。与 *MSRP-FT* 不同，*LEFT-RS* 在 m处保持固定数量的远程延迟单元，从而确保其远程阻塞不会随着故障数量增加而增长。

为了直观展示所提出方法的有效性，我们用与示例 1 相同的设置说明以下示例，该示例也展示了 τ_2 的最坏情况资源访问时间。

示例3。 如图 5所示， τ_1 在 $n_1^x = 6$ 上执行 λ_1，而 τ_2 在 $n_2^x = 1$ 上执行 λ_2。在时间 $t = 1$，两个任务同时请求访问 r^x。在 *LEFT-RS* 下，两个任务开始执行各自的临界区

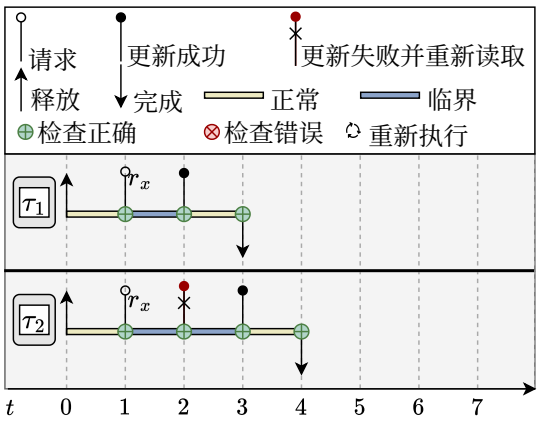


图 5：使用 LEFT-RS 时的 τ_2 最坏情况资源访问

并发执行（规则 *1*）。它们均在$t=2$成功完成，其中 τ_1 更新了 r^x，而 τ_2 被阻塞（规则 *3*）。这代表了 τ_2的最坏情况，因为如果 r^x失败，τ_1 它本应更新 $t=2$。在 τ_2，r^x 重新读取 r^x 并重新执行（规则*4*）。它在 r^x 时完成对 $t=3$ 的访问，并于 $t=4$完成，这早于示例 *1* 中的完成时间。

此外，推论 4 和示例 3 基于 [4]中介绍的 MSRP-FT 理论最坏响应时间分析。MSRP-FT 相关的不可避免开销（如公式（2）所述）尚未纳入考虑。整体性能比较将在第 VII节中展示。

VI. 可调度性分析

本节推导 LEFT-RS 的最坏情况响应时间分析。该分析专用的符号列于表II中，而前文沿用的一般符号可见表 I。

如公式 (3) 所示，R_i 表示给定任务 τ_i 的最坏响应时间。C_i 表示 τ_i 不访问任何共享资源时的纯最坏执行时间。E_i 表示 τ_i 以及优先级高于 τ_i （$lhp(i)$）的本地任务的资源访问时间总和。B_i 表示 τ_i 在到达时遭受的到达阻塞。F_i 表示 τ_i 用于容忍故障的时间。术语 $\lceil R_i/T_h \rceil$表示本地高优先级任务 $\tau_h \in lhp(i)$ 在 R_i期间可以抢占 τ_i 的最大次数。C_h 和 F_h 分别表示 τ_h 每个释放实例的纯最坏执行时间和容错时间。

$$R_i = C_i + E_i + B_i + F_i + \sum_{\tau_h \in lhp(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot (C_h + F_h) \quad (3)$$

我们首先介绍如何界定 E_i，其中包括 τ_i 的资源访问时间，以及 $lhp(i)$ 中任务的资源访问时间；这是因为在最坏情况下，$lhp(i)$ 中的任务会抢占 τ_i，并在抢占期间完成资源访问，从而间接延迟 τ_i [5]。如公式（4）所示，E_i 会遍历系统中的所有共享资源（$\mathcal{R}$）。对于给定资源 r^x，$N^x_{i,\,local}$表示来自 τ_i 以及$lhp(i)$ 中任务对 r^x 发出的请求总数，这些请求称为本地请求。这里我们假设，每个本地请求的执行时间为 $n^x_i = 1$

表 II：可调度性分析符号表

符号	描述
R_i	τ_i的最坏响应时间
E_i	τ_i 和 $lhp(i)$ 的总资源访问时间
B_i	τ_i产生的到达阻塞
F_i, F_h	τ_i 和 $\tau_h \in lhp(i)$的容错时间
$lhp(i)$, $llp(i)$	相对于 τ_i 所在的高/低优先级任务集合同一核心
$\lambda(\tau_i)$	τ_i 所分配到的核心
$\Gamma(\lambda_k)$	分配给核心 λ_k 的任务
$N^x_{i,local}$	τ_i 和 $lhp(i)$ 向 r^x发出的请求数量
$\eta^x_i(R_i,\ R_j)$	τ_j 在周期 R_i 内向 r^x 发出的请求列表以及 R_j
ξ^x_{i,λ_k}	能够阻塞 τ_i 和 $lhp(i)$的远程请求（n^x_j）来自 λ_k
$\mathcal{N}^x_{i,\lambda_k}$	阻塞 τ_i的远程请求的最大数量以及 $lhp(i)$，来自 λ_k
$\mathcal{E}^x_i$	可能造成阻塞的远程请求 n^x_j 总列表 τ_i 和 $lhp(i)$
Syn^x_i	本地重新请求所产生的同步开销对 r^x 的请求
α^x_i	对于 r^x，从 $llp(i)$ 中取最大的 n^x_j
β^x_i	能够阻塞 α^x_i 的远程 n^x_j 列表
$\mathbb{I}(\cdot)$	当条件成立时返回 1 的指示函数。内部成立则返回 1，否则返回 0 的指示函数
$F^A(i)$	对 τ_i 造成到达阻塞的资源
$F(\tau_i)$	被 τ_i访问的资源

因为最坏情况容错时间已计入 F_i和 F_h。$N^x_{i,\,local}$ 在公式（5）中进一步展开。$|\mathcal{E}^x_i|$ 表示本地请求引起的远程阻塞请求总数，具体过程见公式（6）、（7）和（8）中的步骤。Syn^x_i 表示本地请求所经历的同步开销，如公式（9）所述。

$$E_i = \sum_{r^x \in \mathcal{R}} (N^x_{i,local} + |\mathcal{E}^x_i| + Syn^x_i) \cdot c^x \quad (4)$$

如公式（5）所示，$N^x_{i,\,local}$ 包含从 τ_i 到 r^x （N^x_i）的请求总数。在 τ_i 的释放时间内（记为 R_i），τ_i 可能会被 $lhp(i)$ 中的任务抢占；此外，它还通过纳入每个 τ_h 的相应 N^x_h，计入所有 $\tau_h \in lhp(i)$ 的请求。

$$N^x_{i,local} = N^x_i + \sum_{\tau_h \in lhp(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot N^x_h \quad (5)$$

$|\mathcal{E}^x_i|$ 表示 $\mathcal{E}^x_i$ 的大小，该最坏情况列表包含可能阻塞本地请求的远程请求（按 n^x_j 计）。在公式（6）中，$\lambda(\tau_i)$ 表示分配给 τ_i 的核心。$\mathcal{E}^x_i$遍历 τ_i 的所有远程核心（即 $\lambda_k \in \Lambda$，$\lambda_k \neq \lambda(\tau_i)$），并从$\xi^x_{i,\,\lambda_k}$中取前$\mathcal{N}^x_{i,\,\lambda_k}$个请求。

$$\mathcal{E}^x_i = \bigcup_{\lambda_k \in \Lambda, \lambda_k \neq \lambda(\tau_i)} \ \bigcup_{p=1}^{\mathcal{N}^x_{i,\lambda_k}} \xi^x_{i,\lambda_k}(p) \quad (6)$$

$\xi^x_{i,\,\lambda_k}$ 如方程（7）所示，表示关于 n^x_j 来自给定远程核的λ_k 的总请求数的非递增列表。它遍历 λ_k 上的任务（记为 $\Gamma(\lambda_k)$），并获取每个请求的n^x_j以 r^x。记号 $\eta^x_j(R_i,\ R_j)$ 表示

任务在周期 R_i 和 R_j 期间从远程核λ_k 执行时发出的 n_j^x 请求列表，其中计入了背靠背命中 [5]。由此可得，$|\eta_j^x(R_i, R_j)| = \lceil (R_i + R_j)/T_j \rceil \cdot N_j^x$，其中 $|\cdot|$ 表示列表的大小（即请求数），$\lceil (R_i + R_j)/T_j \rceil$ 表示τ_j被释放的次数，而 N_j^x 是在单次释放中τ_j 访问 r^x 的次数。例如，$\eta_j^x(R_i, R_j) = \{2, 2, 2\}$ 表示有三个请求，其 $N_j^x = 3$ 为 $n_j^x = 2$，且所有请求的执行次数均为 $n_j^x = 2$。

$$\xi_{i,\lambda_k}^x = \bigcup_{\tau_j \in \Gamma(\lambda_k), \lambda_k \neq \lambda(\tau_i)} \eta_j^x(R_i, R_j) \tag{7}$$

由于每个核心一次只能处理一个请求，因此每个本地请求在先进先出队列中至多有一个来自远程核的请求排在其前面 [5]。因此，能够阻塞本地请求的请求数量的最大值表示为 λ_k，如公式（8）所示，其等于本地请求数量与给定远程核上的远程请求数量两者中的较小值 λ_k。由于ξ_{i,λ_k}^x 是一个非递增列表，因此前$\mathcal{N}_{i,\lambda_k}^x$ 项是最大的 n_j^x，根据引理1，这可能导致最坏情况同步开销。

$$\mathcal{N}_{i,\lambda_k}^x = Min\{N_{i,local}^x, |\xi_{i,\lambda_k}^x|\} \tag{8}$$

根据引理1，如果所有前置请求都具有 $n_j^x = 1$，则请求不会产生同步开销。必须计入的本地请求同步开销汇总于公式（9）中，其中 $\mathbb{I}(\cdot)$ 是一个指示函数：当括号内的条件成立时返回 1，否则返回 0。公式中的第一项表示具有 $\mathcal{E}_i^x(p) > 1$的前置请求总数。第二项$N_{i,local}^x$表示可能的最大同步开销，因为根据引理3，每个本地请求至多承受一次同步开销。总体同步开销随后取这两个数量中的较小值。

$$Syn_i^x = Min\left\{\sum_{p=1}^{|\mathcal{E}_i^x|} \mathbb{I}\left(\mathcal{E}_i^x(p) > 1\right), N_{i,local}^x\right\} \tag{9}$$

到达阻塞表示为 B_i，如公式（10）所示。由于到达阻塞只能在 τ_i 到达时因本地低优先级任务 $\tau_l \in llp(i)$ 访问优先级高于 τ_i 的上限优先级共享资源，或该资源为全局资源（非抢占式）时发生。能够对 τ_i施加到达阻塞的资源集合表示为 $F^A(i)$。然后，B_i 遍历$F^A(i)$ 中的所有资源，并找出最大的到达阻塞。

$$B_i = \max_{r^x \in F^A(i)} \left\{(\alpha_i^x + |\beta_i^x| + \mathbb{I}(\exists n_j^x \in \beta_i^x,\ n_j^x > 1)) \cdot c^x\right\} \tag{10}$$

第一项 α_i^x 在公式（10）中展开为公式（11），该公式表示来自 n_l^x 的请求中的最大值本地低优先级任务的请求（$\tau_l \in llp(i)$）对给定资源 $r^x \in F^A(i)$（表示为 $N_l^x > 0$）。

$$\alpha_i^x = \max\{n_l^x | \tau_l \in llp(i) \wedge N_l^x > 0\} \tag{11}$$

第二项表示 β_i^x 的大小，它表示由 α_i^x 在公式 12中所产生的远程阻塞请求列表。β_i^x 遍历所有远程核心（$\lambda_k \in \Lambda$，$\lambda_k \neq \lambda(\tau_i)$），并取出$\mathcal{N}_{i,\lambda_k}^x + 1$来自 ξ_{i,λ_k}^x的请求（若长度满足条件（$\mathcal{N}_{i,\lambda_k}^x + 1 \leq |\xi_{i,\lambda_k}^x|$））。这是因为第一个 $\mathcal{N}_{i,\lambda_k}^x$已通过公式（$E_i$（6）取得。

$$\beta_i^x = \bigcup_{\lambda_k \in \Lambda, \lambda_k \neq \lambda(\tau_i)} \left\{\xi_{i,\lambda_k}^x(\mathcal{N}_{i,\lambda_k}^x + 1) \mid \mathcal{N}_{i,\lambda_k}^x + 1 \leq |\xi_{i,\lambda_k}^x|\right\} \tag{12}$$

最后，$\mathbb{I}(\exists n_j^x \in \beta_i^x,\ n_j^x > 1)$ 会检查 β_i^x 中是否存在任何请求满足 $n_j^x > 1$。如果满足，函数返回 1，表示必须按照前文所述增加额外的同步开销；否则返回 0。

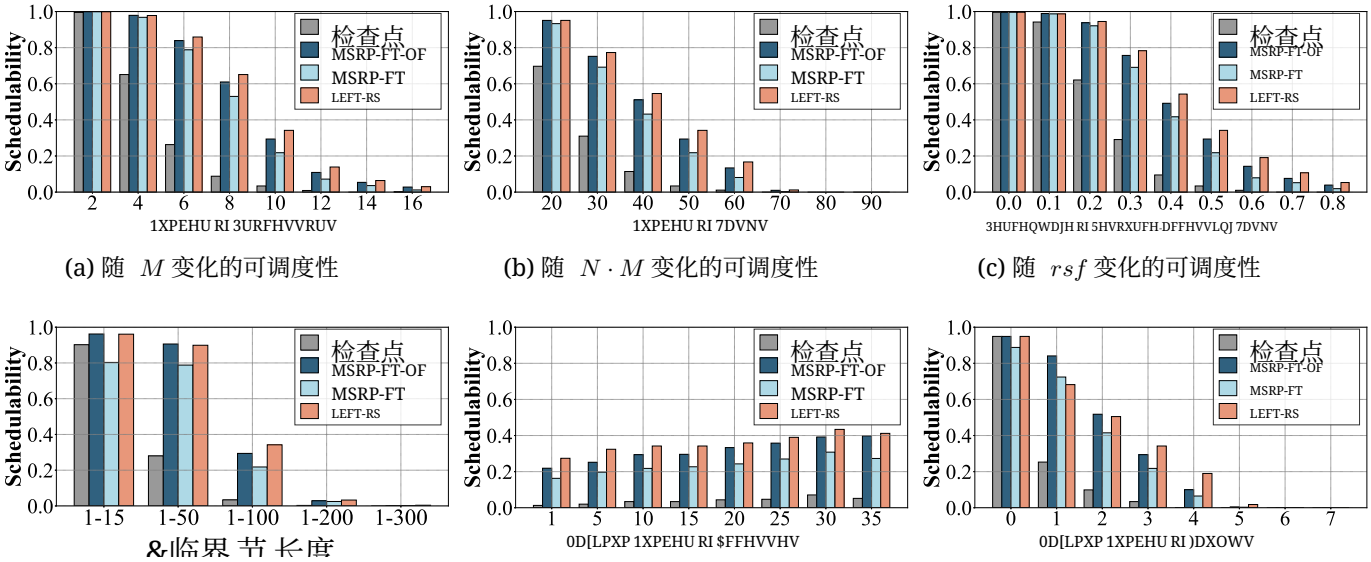
由于检查点将任务划分为多个片段，并且假设每个任务在单次释放的执行过程中最多发生 f_i 次故障，因此最坏情况容错场景是所有 f_i 次故障反复发生在任务中最长的片段上 [19]。因此，τ_i 和 $lhp(i)$ 的最坏情况容错时间如方程（13）所示。该方程通过取普通区段的总执行时间 C_i 与 c^x之间的最大值来确定最长片段；其中，τ_i 所访问的资源中最长的临界区记为 $F(\tau_i)$。最坏情况容错时间的计算方法是将 f_i 乘以最长片段的时长，因为在 LEFT-RS 下，每个任务遇到故障时都会自行重新执行。

$$F_i = f_i \cdot \max\{C_i, \max\{c^x | r^x \in F(\tau_i)\}\} \tag{13}$$

VII. 评估

本节通过比较采用 LEFT-RS、MSRP-FT、MSRP-FT-OF（无开销）和检查点机制的系统的可调度性，评估所提出方法的有效性。所有方法均采用相同的检查点放置策略，从而确保公平比较，且不考虑检查点开销。它们的差异仅在于发生故障时如何管理全局资源共享。MSRP-FT 和 MSRP-FT-OF 均采用第III-B节中介绍的协助机制。具体而言，MSRP-FT-OF 不考虑协议开销，而 MSRP-FT 则计入公式（2）所定义的开销。相比之下，检查点机制要求任务在持有锁时发生故障后仅按顺序重新执行。

我们通过生成合成测试来进行评估。我们考虑包含一组核心 $M = [2, 4, 6, 8, 10, 12, 14, 16]$ 的多核平台。对于每个核心，我们将每个核心的任务数量设置在 $N = [2, 3, 4, 5, 6, 7, 8, 9]$ 范围内，从而为可调度系统的分析和比较提供适当的范围。每个任务的利用率 U_i 通过 UUnifast 算法 [20],生成，总系统利用率上限为 $U_{\text{total}} = 0.04 \cdot M \cdot N$。任务周期是随机



(a) 随 M 变化的可调度性 (b) 随 $N \cdot M$ 变化的可调度性 (c) 随 rsf 变化的可调度性

(d) 随 L 变化的可调度性 (e) 随 A变化的可调度性 (f) 随 f 变化的可调度性

图 6：具有 $N = 5$、 $M = 10$、 $A = 10$、 $L = [1, 100]$、 $rsf = 0.5$、 $f = 3$ 和 $K = M$ 个资源时系统的可调度性

在对数均匀分布中从 [1毫秒到1000毫秒]之间选择。任务的截止时间等于其周期。对于任务τ_i，总最坏情况执行时间（包括临界区）“由 $C_i = U_i \times T_i$给出。

每个系统包含 K 个共享资源，数量等于核心数（即 $K = M$）。在生成的任务中，选择一定比例的任务来请求随机数量的资源（最多 K个）。该比例以资源共享因子 $rsf \in [0, 0.8]$表示。每个任务在单次释放期间对某一资源的最大访问次数设为 $A = [1, 5, 10, 15, 20, 25, 30, 35]$。共享资源的最坏情况计算时间在 $L \in [1\mu s, 300\mu s]$范围内随机生成。我们将 $\frown$个 τ_i 临界区的总长度记为 C_i^r，然后令 $C_i = C_i - C_i^r \geq 0$以得到 C_i。我们将任务在单次释放期间可能遭遇的最大故障数设为 $f \in [0, 7]$。例如，若$f = 5$，则任务在一次释放中可能随机遭遇 0 至 5 次故障。此设置采用了[4] 中使用的相同故障模型，以确保比较公平。这一设定有充分依据：辐射和电磁干扰研究证实，单个高能粒子或干扰可能引发多位翻转或多节点翻转，因此，在安全关键嵌入式系统中，每个任务发生多次瞬态故障是现实且重要的情形 [21]。此外，任务采用最差适配启发式算法进行分配，并使用 FP-FPS 进行调度。采用截止时间单调优先级排序。

为了量化 MSRP-FT 引入的结构性协调开销（如限制 2 所述），我们依赖于在多核平台上部署的实时操作系统的实证测量结果。封装构造开销，记为 O_{wrap} 和 $O_{\text{self wrap}}$，见公式（2），源于发布共享描述符的成本，该描述符包含函数指针、资源引用和同步标志。底层操作包括内存设置、全局可见性强制和内存屏障。Shi 等人 [16] 报告称，在 LITMUS$^{\text{RT}}$ 中启用协助机制所产生的封装设置成本始终低于 $1\,\mu s$。类似地，一个基于 Linux 的容错消息传递原型报告称，跨核消息发布延迟约为 $0.55\,\mu s$ [18]。基于这些证据，我们在评估中保守地将 $O_{\text{wrap}} = O_{\text{self 封装}} = 1\,\mu s$ 设为该值。

每副本执行开销，O_{replica} 表示辅助者开始执行前产生的成本：轮询以确认就绪、获取操作描述符以及复制资源状态。Zhao 等人 [17] 测量了 LITMUS$^{\text{RT}}$ 上 MrsP 的协作执行延迟，约为 $8.4\,\mu s$。Shi 等人 [16] 进一步将其分解为迁移本身的开销 $5.6\,\mu s$，以及辅助核心上上下文重新进入的开销 $1.5\,\mu s$。即使是商用实时操作系统平台，其数量级也相近，例如，VxWorks 的开销为 $11\,\mu s$，RTLinux 的开销为 $13.4\,\mu s$ [22]。基于这些结果，我们设定 $O_{\text{replica}} = 6\,\mu s$，这反映了多核嵌入式系统中辅助者驱动的协调与执行的现实典型成本，从而避免与 LEFT-RS 相比时夸大开销。

在以下评估中，对于每种系统设置组合，均生成 1000 个系统，并展示所考虑模型中可调度系统的百分比。图 6中，y 轴表示在生成的 1000 个系统中可调度系统所占的比例（记为可调度性）。

观察 1： 如图 *6*所示，在所有设置下，*LEFT-RS* 的性能始终优于检查点方法。

由于这两种方法仅在全局资源的容错机制上有所不同，因此这一性能差距验证了，直接集成传统容错技术会导致难以管理的全局资源竞争。尽管检查点通过将任务划分为片段降低了重新执行成本

，但它并不能缓解容错全局资源访问中的竞争。发生故障时，顺序重新执行导致的锁持有时间延长，可能会显著延误其他任务，造成系统效率低下。

观察 2： 在图6a、6b、6c、6d 和 6e中，*LEFT-RS* 方法的表现始终优于 *MSRP-FT*。

如图6a所示，在每个核心的任务数固定为 $N = 5$时，将核心数从 2 增加到 16 会降低所有方法的可调度性。这种下降主要是因为随着核心和任务数量增加、竞争访问全局资源的情况加剧。

随着核心数增加，LEFT-RS 与 MSRP-FT 之间的性能差距也随之扩大。例如，在 4 个核心的情况下，LEFT-RS 成功调度的系统数量比 MSRP-FT 多 10 个；而在 6 个核心的情况下，这一差距扩大到 71 个系统。这表明，随着全局资源竞争加剧，LEFT-RS 能更有效地管理资源访问和容错。尽管 MSRP-FT 采用容错机制，通过利用自旋任务并行执行临界区并加快恢复，但它仍存在一些根本性的局限。如局限性 1 和 2 （第 III-C节）中所述，MSRP-FT 中的任务必须按顺序管理资源访问。如果任务遇到故障，就会延迟先进先出队列中的所有后续任务。此外，如公式（2）所示，MSRP-FT 特有的开销会直接受到来自不同核心的请求数量影响。相比之下，LEFT-RS 允许任务独立执行临界区，无需遵循先进先出顺序；这不仅使任务在成功更新后能够退出先进先出队列，也免除了与 MSRP-FT 相关的开销。总体而言，如图 6a 所示，LEFT-RS 的表现平均优于 MSRP-FT 51.3%。

在图6b、6c和6d中也观察到了类似趋势：总任务数量、共享资源的任务比例以及临界区长度等系统参数的变化，会导致全局资源竞争加剧。在所有这些情形下，LEFT-RS 的表现始终优于 MSRP-FT。具体而言，在图6b、6c和6d中，LEFT-RS 分别实现了平均 62.9%、58.8% 和 84.5% 的性能提升。此外，当 $L = [1\mu s, 15\mu s]$ 在图6d中时，检查点优于 MSRP-FT，这表明临界区较小时，按顺序重新执行比采用协助机制更具成本效益。

如图6e所示，资源访问最大次数增加并不一定会降低可调度性。这是因为，在系统利用率约束下，访问次数增加有时会缩短所生成共享资源的临界区长度。在 [4], [12] 中也观察到了类似影响。尽管如此，LEFT-RS 的表现仍优于 MSRP-FT，平均提升 53%。

观察3： 当系统中的故障数量非常少时，*LEFT-RS* 相较于 *MSRP- FT* 的优势会消失。

如图 6f所示，当 $f = 0$时，根据定义，检查点的表现与传统 MSRP完全相同，因为系统中不考虑故障。LEFT-RS 的表现也完全相同

表 III：仅 MSRP- FT（左）或仅 LEFT-RS（右）可调度的系统数量

A	MSRP-FT √ LEFT-RS	LEFT-RS √ MSRP-FT		f	MSRP-FT √ LEFT-RS	LEFT-RS √ MSRP-FT
1	0	111		0	0	61
5	0	127		1	52	10
10	0	124		2	0	90
15	0	115		3	0	124
20	0	116		4	0	125
25	1	121		5	0	15
30	0	126		6	0	0
35	0	139		7	0	0

于检查点，因为根据第 10 条规则，避免了同步开销（见第IV节）。MSRP-FT 的额外开销使其表现逊于其他协议。当每个任务的最大故障数被随机设定为不超过 1 时，由于大多数资源访问都不会发生故障，LEFT-RS 的部分优势不及 MSRP-FT，平均差距为 6.2%。这一行为符合推论 4的预测：由于故障罕见，每个远程请求占据资源持有者位置的时间都不长；在这种情况下，与临界区长度相比，MSRP-FT 的开销是可控的。

然而，随着 f 增加，LEFT-RS 的优势便显现出来。在 LEFT-RS 具有优势的情形下（除 $f = 1$外），其性能平均优于 MSRP-FT 110%。在 MSRP-FT 下，资源访问遵循一次服务一个任务的语义，即队首任务占据独占服务槽。如果该任务在临界区内频繁发生故障，就会延长先进先出队列中后续任务的阻塞时间。相比之下，LEFT-RS 将每个请求的最大资源访问时间单位限制为 $n_i^x + m + 1$，如推论 2和4所示，且不受请求遭遇的故障次数影响。这可以防止过多故障损害系统性能。

表III 比较了只能由 MSRP-FT 或 LEFT-RS 调度的系统数量。表格左半部分改变参数 A，对应图6e中的设置。本部分下的第一列（标记为“MSRP-FT ✓/ LEFT-RS ✗”）显示只能由 MSRP-FT 调度、而不能由 LEFT-RS 调度的系统数量。第二列（标记为“LEFT-RS ✓/MSRP-FT ✗”）则表示相反的情况。同样，表格右半部分改变参数 f，如图6f所示。该表突显了两种方法在不同参数配置下各自互补的优势。

观察 4： *LEFT-RS* 不仅在整体可调度性方面优于 *MSRP-FT*，在 *MSRP-FT* 难以应对的场景中也是如此。

当变化 A时，如表 III所示，LEFT-RS 始终在单向上优于 MSRP- FT。具体而言，它几乎能够调度所有在 MSRP-FT 下被认为可调度的系统，并且还能处理大量在 MSRP- FT 下无法调度的系统。当 $A = 25$时，只有一个系统仅能由 MSRP-FT 调度。这与推论 4一致，该推论指出

我们在理论上并非在所有场景下都优于 MSRP-FT。在此情况下，$f = 3$（最大故障数）的设置使生成的任务可能遭遇 0 到 3 次故障。MSRP-FT 偶尔占优的情形出现在任务遭遇的故障较少时，此时相关开销相对于临界区长度仍处于可控范围内。

当变化容错级别 f 时，从表格右侧也可以得出类似的观察结果。当 $f = 1$ 时，MSRP-FT 成功调度了更多无法由 LEFT-RS 调度的系统。这是意料之中的，因为这些配置中的许多任务不存在故障，或仅经历一次故障，这限制了 LEFT-RS 的容错设计。即便如此，在 1000 个系统中，LEFT-RS 也仅漏掉了 52 个。当f 增加到 2 时，这一数字立即降至 0。随着f 进一步增加，LEFT-RS 的优势愈发明显。这些发现进一步证明了该方法在日益增强的容错场景下的可靠性和有效性。

观察5： *LEFT-RS* 相较于 *MSRP-FT-OF* 保持着明显优势。

由于 MSRP-FT-OF 本质上伴随着 LEFT-RS 所没有的外部开销，因此将二者进行比较并无实际意义。尽管如此，结果仍具有参考价值。如图 6所示，总体可调度性趋势与 MSRP-FT 的情况相同：各柱之间的相对差异得以保留，但与 LEFT-RS 相比的性能差距有所缩小。尽管 MSRP-FT- OF 在大多数情况下仍劣于 LEFT-RS，但在故障数量较低时（见图6f），还出现了一个额外实例（$f = 2$），其中 MSRP-FT-OF 的可调度性略高于 LEFT-RS。这一行为与观察3一致。最后，对 MSRP-FT-OF 的展示证实，LEFT-RS 的优势源于其协议设计，而不仅仅是因为排除了开销。

VIII. 结论

本文提出了一种面向多核实时系统的无锁容错资源共享（LEFT-RS）协议。与 MSRP-FT 不同，后者的并发仅服务于队首请求，访问仍然是串行的；LEFT-RS 通过允许每个请求在并发读取后于本地执行，实现了真正的并发。成功执行后，任务可以更早完成，从而在瞬态故障频发时显著减少阻塞。独立执行避免了大量协调开销，并增强了故障恢复能力。我们还提出了全面的最坏情况响应时间分析，以保证时序可预测性。评估结果表明，我们的方法始终优于最先进的协议，平均可调度性最高提升 84.5%。未来的扩展将不再局限于非抢占式执行，并采用更先进的调度策略，例如基于优先级上限的协议。

IX. 致谢

本研究部分由 InnovateUKSCHEME 项目（10065634）资助。EPSRC 研究数据管理：本研究期间未创建任何新的原始数据。

参考文献

[1] B. B. Brandenburg，《多处理器实时锁协议》，载于《实时计算手册》。Springer，2022 年，第 347–446 页。[2] T. Vijaykumar、I. Pomeranz 和 K. Cheng，“使用同时多线程进行瞬态故障恢复”，《ACM SIGARCH 计算机体系结构新闻》，第 30 卷，第 2 期，第 87–98 页，2002 年。[3] L. Osinski、T. Langer 和 J. Mottok，“不同体系结构层级上的容错方法综述”，载于第 30 届计算机系统体系结构国际会议（ARCS），2017 年，第 1–9 页。[4] N. Chen、S. Zhao、I. Gray、A. Burns、S. Ji 和 W. Chang，“MSRP-FT：多处理器混合关键度系统中的可靠资源共享”，载于 IEEE 第 28 届实时与嵌入式技术及应用研讨会（RTAS），2022 年，第 201–213 页。[5] S. Zhao、J. Garrido、A. Burns 和 A. Wellings，“MrsP 的新可调度性分析”，载于 IEEE 第 23 届嵌入式与实时计算系统及应用国际会议（RTCSA），2017 年，第 1–10 页。[6] V. Izosimov、P. Pop、P. Eles 和 Z. Peng，“时间与成本约束的容错分布式嵌入式系统设计优化”，载于 IEEE欧洲设计、自动化与测试会议（DATE），2005 年，第 864–869 页。[7] N. Miskov-Zivanov、K.-C. Wu 和 D. Marculescu，“感知工艺变异的瞬态故障建模与分析”，载于 IEEE/ACM 计算机辅助设计国际会议，2008 年，第 685–690 页。[8] P. Gai、M. Di Natale、G. Lipari、A. Ferrari、C. Gabellini 和 P. Marceca，“在 Janus 片上多处理器平台上共享资源时 MPCP 与 MSRP 的比较”，载于第 9 届 IEEE实时与嵌入式技术及应用研讨会（RTAS），2003 年，第 189–198 页。[9] A. Burns 和 A. J. Wellings，“与可调度性兼容的多处理器资源共享协议——MrsP”，载于 IEEE 第 25 届实时系统欧洲微型会议（ECRTS），2013 年，第 282–291 页。[10] A. Block、H. Leontyev、B. B. Brandenburg 和 J. H. Anderson，“一种适用于多处理器的灵活实时锁协议”，载于第 13 届 IEEE 嵌入式与实时计算系统及应用国际会议（RTCSA），2007 年，第 47–56 页。[11] B. B. Brandenburg 和 J. H. Anderson，“最优多处理器实时锁协议的 OMLP 系列”，《嵌入式系统设计自动化》，第 17 卷，第 2 期，第 277–342 页，2013 年。[12] S. Zhao、H. Xu、N. Chen、R. Su 和 W. Chang，“FRAP：一种适用于多处理器实时系统的灵活资源访问协议”，载于 IEEE实时系统研讨会（RTSS），2024 年，第 349–361 页。[13] J. H. Anderson、S. Ramamurthy 和 K. Jeffay，“使用无锁共享对象进行实时计算”，ACM《计算机系统汇刊》（TOCS），第 15 卷，第 2 期，第 134–165 页，1997 年。[14] S. S. Nabavi 和 H. Farbeh，“一种适用于多处理器实时系统的容错资源锁协议”，《微电子学杂志》，第 137 卷，文章编号 105809，2023 年。[15] P. Gai、G. Lipari 和 M. Di Natale，“在单处理器和多处理器片上系统中，最小化实时任务集的内存利用率”，载于第 22 届 IEEE实时系统研讨会（RTSS），2001 年，第 73–83 页。[16] J. Shi、K.-H. Chen、S. Zhao、W.-H. Huang、J.-J. Chen 和 A. Wellings，“在 LITMUS^RT 上实现并评估多处理器资源同步协议（MrsP）”，载于第 13 届嵌入式实时应用操作系统平台研讨会，2017 年。[17] S. Zhao、J. Garrido、R. Wei、A. Burns、A. Wellings 和 J. A. de la Puente，“针对嵌套资源的 MrsP 完整运行时开销感知可调度性分析”，《系统与软件杂志》，第 159 卷，文章编号 110449，2020 年。[18] G. Losa、A. Barbalace、Y. Wen、H.-R. Chuang、B. Ravindran 和 M. Sadini，“在商用多核硬件上通过机内全软件栈复制实现透明容错”，载于 IEEE 第 37 届分布式计算系统国际会议（ICDCS），2017 年，第 1521–1531 页。[19] S. Punnekkat、A. Burns 和 R. Davis，“实时系统检查点分析”，《实时系统》，第 20 卷，第 83–102 页，2001 年。[20] E. Bini 和 G. C. Buttazzo，“测量可调度性测试的性能”，《实时系统》，第 30 卷，第 1 期，第 129–154 页，2005 年。[21] R. C. Baumann，“先进半导体技术中的辐射诱发软错误”，IEEE《器件与材料可靠性汇刊》，第 5 卷，第 3 期，第 305–316 页，2005 年。[22] B. Ip，“VxWorks 与 RTLinux 的性能分析”，计算机科学系嵌入式系统语言部门，2001 年。