机会互斥

Karthi Srinivasan电气工程系
耶鲁大学纽黑文，美国
karthi.srinivasan@yale.edu

Yoram Moses电气工程系 以色列理工学院海法，以色列
moses@ee.technion.ac.il

Rajit Manohar电气工程系 耶鲁大学纽黑文，美国
rajit.manohar@yale.edu

arXiv:2305.05802v1 [cs.DC] 9 May 2023

摘要—在共享资源使用中，互斥是一个重要问题：在任意给定时刻，只能有一个进程使用共享资源。如果互斥协议不使用各进程占用资源时长的信息，可能导致资源利用时间不够理想。我们研究一个简单的双进程互斥问题，其中中央服务器负责提供对共享资源的访问。我们表明，即使没有时钟，在满足某些条件时，服务器也可以根据时序信息，机会性地提前授予客户端访问权限。我们将这一新协议称为机会互斥。我们的方法要求客户端与服务器之间的每条信道都额外配备一个请求信号，以传递额外信息；服务器仅根据事件发生的顺序，而非通过测量时间，就能提前授予访问权限。我们推导了该协议的握手规范和生产规则，并报告了采用65纳米工艺制造的电路的能耗和延迟。

索引词—**互斥、仲裁、时序、异**

I. 引言

互斥问题——在多个相互竞争、各自请求访问某个共享资源的进程之间，保证对该资源的访问互斥——早已为人所知，已有多种算法在不同模型下解决这一问题 [1]–[3]。

传统上，互斥问题通过实例化一个持有令牌的中央服务器来解决，该令牌用于授予对共享资源的访问权。任何客户端请求使用资源时，服务器都会将令牌交给该客户端；客户端随后使用资源，并在使用完毕后将令牌交还服务器。如此循环往复。由于任何时刻都只有一个客户端可以持有令牌，因此能够保证互斥。该问题也存在分布式解决方案，例如使用环 [4], [5]，或进程树 [6]。

通常，在设计负责处理整个系统的互斥服务器时，不会对请求进程的行为作任何假设。然而，在异步电路领域，时序分析和时序仿真用于在设计流程中确定性能[7], [8]。这些测量结果也有助于进一步优化设计。本文探讨了基于对请求进程时序行为的了解来优化互斥过程的可能性，并提出了一种实现这一目标的新方法。

我们考察一个简单情形：一个单令牌服务器和两个客户端请求使用共享资源。我们证明如果服务器事先知道客户端时序行为的某些界限，那么即使没有内部时钟，它也可以仅根据客户端信号到达的顺序关系，在另一个客户端仍在使用资源时，抢先授予一个客户端访问权限，从而减少共享资源的空闲时间，且减少量可能没有上限。

在以下各节中，我们将详细介绍在设计机会互斥电路时所使用的各种时序约束，并展示如何将这些特定约束整合到电路本身中。随后，我们推导出一种使用三个仲裁器的直接实现方案。接下来，我们展示另一种看似使用两个仲裁器、但实际上可以简化为单个仲裁器的实现方案。最后，我们给出 SPICE 仿真结果，并讨论该电路的潜在用途。

II. 时序分叉与之字形

在电路设计中，可以利用同时引发两个事件的共同事件，推断这两个事件之间的顺序关系。在某些情况下，即使两个不同进程之间并无实际通信，也能推断出这两个进程上的两个事件的时间顺序；这种情况尤其值得关注。这类约束称为分歧点约束。同步逻辑设计中的建立时间约束就是一个常见示例：数据必须在时钟沿到达之前至少一段规定的时间有效。

图 1 展示了这种分歧点约束的一个简单模型，称为时序分叉。假设 A、B 和 C 是三个进程，其中 B 中的一个事件 e_1 直接或间接地导致 A 和 C 上分别发生事件 e_3 （称为头部）和 e_2 （称为尾部）。设事件 e_i 的发生时间为 t_i。假设 t_3 与 t_1 之间的延迟为d_A，而 t_2 与 t_1 之间的延迟为 d_C，这两种延迟都可能落在某个范围内。现在，如果 d_A 始终大于 d_C，我们可以得出结论：

$$\begin{aligned} W &= \inf(d_A - d_C) \\ &= \inf((t_3 - t_1) - (t_2 - t_1)) \\ &= \inf(t_3) - \sup(t_2) > 0 \end{aligned}$$

这种关于两个事件发生时间相对时间差的约束称为时序分叉，而衡量最小时间间隔的量 W

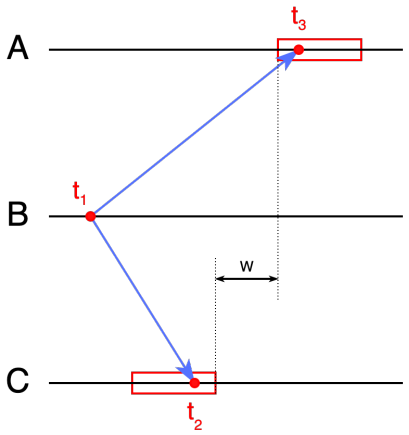


图 1. 时序分叉。一个分歧点约束的简单示例。红色方框表示事件可能发生的时间窗口。红点表示事件的一种特定实现。

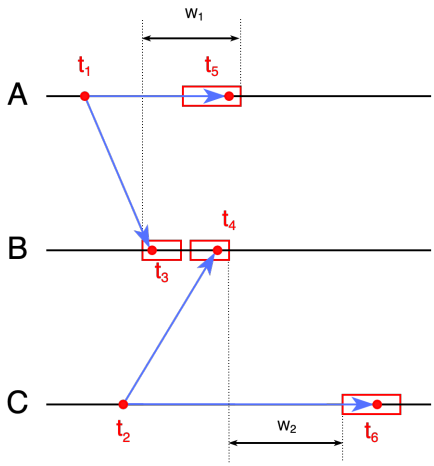


图 2. 时序之字形。将两个时序分叉结合起来，并利用一对事件的顺序关系，推断不共享同一分歧点的事件之间的相对时间。

两个事件之间的间隔称为分叉权重。需要注意的是，“最小间隔”这一理解仅适用于正权重分叉。如果分叉权重为负，则定义保持不变，但其所表示的量的直观含义会转变为“最大间隔”。此外，这三个进程不必互不相同。存在一种退化情形，即 A 和 B，或 C 和 B，属于同一个进程。只要发生时间仍然遵循相同的规律，该约束就依然成立。需要注意的是，实际发生时间可能处于一个取值范围内，但分叉权重是根据这些发生时间的上界和下界计算的。现在，我们可以通过使用多个时序分叉来扩展这一概念，具体如下。

考虑图 2 所示的情况，其中 A、B 和 C 是三个不同的进程。我们希望在缺乏导致事件 e_5 和 e_6 发生的实际事件顺序信息（即 e_1 和 e_2）的情况下，对事件 e_5 和 e_6 的顺序关系作出绝对判定。

假设 e_1 进程 Acauses 两个事件 e_3 和 e_5，它们各自都可能在一个时间范围内发生，使得

$$t_3 - t_5 \geq -W_1 \quad (W_1 > 0)$$

从而形成一个时序分叉。换句话说，两个子事件的发生时间之间的最小间隔有一个下界：

$$\inf(t_3) - \sup(t_5) = -W_1$$

同样，对于进程 C：

$$t_6 - t_4 \geq W_2 > 0, \text{where}$$
$$\inf(t_6) - \sup(t_4) = W_2$$

请注意，这种表述并未要求事件 e_1和 e_5 发生在同一进程中，只要求 e_5 由e_1引起，因此发生在之后的某个时间。图 2 采用这种画法，是为了更好地衔接后续章节。 e_2 也是如此。和 e_6。现在，在一种特定情况下，即当 e_4 发生在 e_3（$t_4 - t_3 \geq 0$）和 $W_2 - W_1 > 0$ 之后时，我们可以如下确定事件 e_5 和 e_6 的顺序关系：

$$\begin{aligned}(t_6 - t_5) - (t_4 - t_3) &\geq W_2 - W_1 \\ \implies (t_6 - t_5) &\geq W_2 - W_1 \\ \implies t_6 - t_5 &> 0\end{aligned}$$

这种时间约束称为时序之字形：通过适当地组合多个时序分叉，可以确定那些并非由单一分歧点连接的事件之间的顺序关系。数量 $W_2 - W_1$ 称为之字形权重。需要理解的关键之处在于，不存在一个共同事件或单一分歧点来确定事件 e_1 和 e_2的时间，而且它们彼此完全独立。换言之，这种情况在本质上不同于简单的时序分叉，无法简化为时序分叉。事实上，在上述分析中，我们甚至没有假设事件 e_1 和 e_2 之间存在顺序关系。尽管如此，我们仍能根据其他时序信息推断出一些时序信息，而这些信息并非简单的“a 先于 b”关系。

此外，这种组合方法可以轻松扩展，从而构造出由任意数量时序分叉组成的之字形。这可以为看似无关的事件集合的时序提供更详细、层次更高的信息。

在以下描述中，据我们所知，我们首次将之字形因果关系 [9], [10]，应用于电路设计。具体而言，我们利用这一方法来解决经典的互斥问题。

III. 机会互斥

在传统的互斥设置中，有一个共享资源，需要确保同一时间最多只能由一个客户端使用；多个客户端会争用该资源的使用权；服务器则持有一个令牌，用来决定谁有权使用。

该资源。服务器会将令牌发放给某个提出请求的客户端，并任意决定发给谁。客户端使用完资源后会归还令牌。服务器接收令牌后，就可以决定接下来将资源分配给哪个客户端。这个进程会不断重复，可能永远持续下去。

考虑图 3 所示的场景，其中 C1 和 C2 是向服务器 S 发出请求以使用共享资源（未显示）的两个进程。在最一般的情况下，我们不假设预先知道每个进程何时会请求使用共享资源，或停止使用并释放共享资源。

在这里描述的场景中，假设服务器掌握以下信息：

1) **提前释放时间**：C1 通知服务器的时间（$t_3 - t_1$），即资源实际停止使用之前的时间。2) **抢占时间**：C2 向服务器发送请求的时间（$t_4 - t_2$），即需要资源之前的时间。3) **链路延迟**：C1、C2 与 S 之间导线上的延迟。

再说一次，上述定义实际上都是时间区间；当我们说服务器掌握这些信息时，意思是它知道这些时间区间的上下界。利用这些信息，S 可以计算 W_1 的上界和 $t_3 - t_{re1}$，以及 W_2 的下界和 $t_4 - t_{r2}$。如果 $W_2 \geq W_1$，那么服务器就能采取一些有意思的操作。我们称之为非对称情况，因为 C2 提前释放、C1 提前请求时刻的互补界限是未知的。如果这些界限已知，那么我们就处于对称情形。

现在，假设 C1 正在使用资源，而 C2 在资源仍被占用时发出请求。如果 C2 的请求如图 3(a) 所示，在提前释放之前到达，那么服务器就无法确保互斥。由于服务器没有内部时钟，因此无法衡量抢占请求到达得有多早。所以，服务器必须等待 C1 实际释放资源，才能确认资源空闲，然后才能批准 C2 使用资源。

如图 4 所示，信道在物理结构上的重要区别是：C1 必须有两根请求线接入服务器，分别是提前请求线 (r_e) 和实际请求线 (r_a)；请求资源时，两根线都会拉高。释放资源时，C1 会先按照上述提前释放时间约束拉低提前请求线，以表示“即将完成”。等到真正用完资源后，C1 再拉低实际请求线。

有趣的情况是，来自 C2 的请求在 C1 提前释放之后才到达。在这种情况下，由于我们知道之字形具有正权重（$W_2 - W_1 \geq 0$），服务器知道，即使立即授予批准，资源最早被使用的时间也晚于资源最晚被释放的时间。因此，如图 3(b) 所示，服务器会授予提前批准。与通常情况下服务器必须等待明确的情况相比，这减少了资源处于空闲状态的时间。

有关资源使用结束的信息，以便访问该资源。请注意，要检查这项提前批准是否合法，服务器无需知道t_{r2} 和 t_{re1} 之间的相对时间，只需知道它们发生的顺序即可，因此无需使用内部时钟。我们将这种方式称为机会式互斥，因为所需的事件顺序只能在运行时确定。如果确实出现了这种特殊情况，服务器就可以“伺机”在撤销对第一个资源的访问权限之前，授予对另一个资源的访问权限，而不会违反实际的互斥约束。

对称情形与此非常相似，两个信道都需要提前请求导线和实际请求导线，如图5所示。要实现这一点，我们需要两个不同的之字形具有正权重。这两个之字形彼此独立。一个将 C1 的提前释放时间与 C2 的抢占时间关联起来，另一个则将 C2 的提前释放时间与 C1 的抢占时间关联起来。实际上，C1 和 C2 各自都有两个独立的时间变量，可以分别独立确定这些变量，因此可能产生零个或多个具有正权重的之字形。在这种情况下，设计者在决定何种情况应被视为抢占式请求时拥有更大的自由度。在非对称情况下，C2 只需拉高一条请求线，就有可能接收到机会式授权。而在这里，服务器可以要求它同时拉高两条导线，才将其视为机会情况；也可以只要求拉高其中一条。在后文介绍的实现中，我们假设采用前一种方式。对于这两种情况，只需增加少量逻辑门即可扩展其电路，从而能够通过一个比特开启或关闭机会模式。

再一次强调，上述行为无法用时序分叉捕捉，因为 C1 和 C2 的行为之间不存在任何耦合；要获得任何有用的时序信息，就需要之字形。

IV. 三仲裁器方法

上一节所述非对称情况的一种直接实现如下所示，只需一个内部状态变量。有关 CHP 表示法的详情，请参阅附录I。

$$
\begin{array}{l}
x = 0; \\
*[[\,|\, C1.r_e \wedge C1.r_a \longrightarrow C1.a\uparrow \\
\quad [] \quad\quad x \vee C2.r \longrightarrow C2.a\uparrow \\
\;\,|\,]; \\
\;\,[\neg C2.a \longrightarrow \\
\quad\;\, [\,|\, \neg C1.r_e \longrightarrow \\
\quad\quad\;\, [\,|\, \neg C1.r_a \longrightarrow C1.a\downarrow \\
\quad\quad\quad [] \quad C2.r \longrightarrow C2.a\uparrow; x\uparrow; [\neg C1.r_a]; C1.a\downarrow \\
\quad\quad\;\, |\,] \\
\quad\quad\;\, [] \quad C2.r \longrightarrow [\neg C1.r_a]; C1.a\downarrow \\
\quad\;\, |\,] \\
\;\,[]\neg C1.a \longrightarrow [\neg C2.r]; C2.a\downarrow; x\downarrow \\
\;\,] \\
\,]
\end{array}
$$

第一次仲裁在资源的空闲状态下从两个请求方中进行选择，并以.....的形式授予批准。

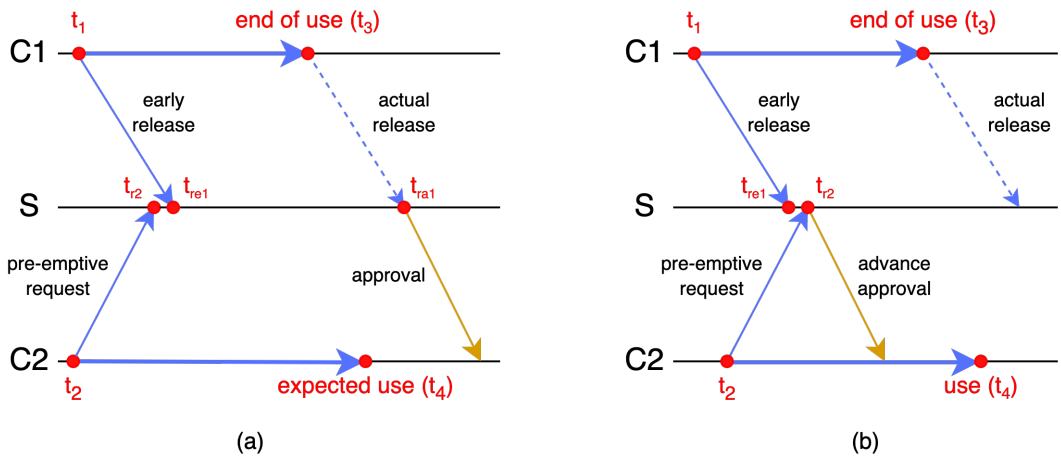


图 3. 时序之字形。（a）标准情况：第二个进程的请求到达得太早，无法利用时序之字形。（b）有趣的情况：第二个进程的请求到达时，介于第一个进程提前释放和实际释放之间。

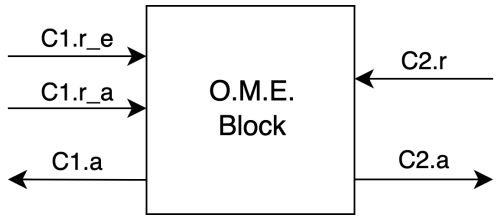


图 4. 机会互斥模块 - 非对称情况

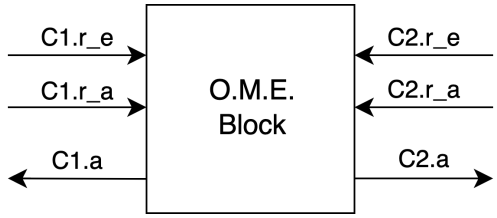


图5. 机会式互斥块 - 对称情形

基于仲裁的确认信号。由于这是非对称情况，如果 C2 获准访问，则无需进行任何有趣的操作，服务器只需等待握手完成，然后返回起点。然而，如果 C1 获准访问，服务器就会进入第二次仲裁。这里有三种情况：两种常规情况和一种有趣的情况。两种常规情况是：

- C2 在提前释放之前的请求。
- C1 在 C2 甚至还未发出请求之前就已完成使用。

在第一种情况下，我们无法利用之字形，因为如果向 C2 授予批准，就有可能违反互斥原则。值得注意的是，服务器在提前释放之前，并不知道 C2 的请求究竟提前了多久，因此无法利用之字形。在第二种情况下，C1 在 C2 发出请求之前就已完成，因此也没有什么值得特别处理的。

最后，电路设计用于利用的有趣情况是：

这种情况发生在：

- C2 在 C1 提前释放与实际释放之间发出的请求。

在这种情况下，每当 C1 执行实际释放时，服务器都会先批准 C2，然后才完成与 C1 的握手。状态变量 x 用于在以这种方式批准 C2 后绕过首次仲裁，以便完成握手。完成后，系统便回到静止状态，进程随之重复。

在此实现中，需要进行三次非确定性选择，因此需要三个仲裁器。这在实现过程中可能会相当昂贵。

接下来，我们将介绍一种减少仲裁器数量的替代方法。最初描述的握手扩展看起来需要两个仲裁器，但我们将证明，它可以简化为只需要一个仲裁器。

V. 单仲裁器方法

在下面的另一种实现中，我们需要两个状态变量，但只需进行两次非确定性选择，而不是三次。第二个进程执行初始仲裁。与之前一样，先批准 C2 不会产生任何有趣的情况。如果批准了 C1，而 C2 请求得太早，那么在继续之前，会先完成与 C1 的握手。第一个进程的作用是在状态变量 g 设定时，与其他操作并行完成和 C1 的握手。如果 C2 在提前释放和实际释放之间发出请求，那么允许与 C1 的握手和批准 C2 并行完成。从本质上说，这是先前所述方法的一种并行形式，其中包含三个仲裁器。

$$
\begin{array}{l}
g = 0; \\
*[[g \wedge \neg C1.r_a \wedge \neg C1.r_e]; C1.a\downarrow; g\downarrow] \\
\| \\
*[[|\neg g \wedge C1.r_e \longrightarrow [C1.r_a]; C1.a\uparrow; \\
\qquad\qquad\qquad\qquad [| \quad C2.r \longrightarrow g\uparrow; [\neg g] \\
\qquad\qquad\qquad\qquad \;[]\neg C1.r_e \longrightarrow g\uparrow \\
\qquad\qquad\qquad\qquad |] \\
\quad [] \qquad C2.r \longrightarrow C2.a\uparrow; [\neg g \wedge \neg C2.r]; C2.a\downarrow \\
\; |] \\
]
\end{array}
$$

现在请注意，我们实际上可以将嵌套的非确定性选择重写为一次 4 路非确定性选择，并引入一个新的状态变量，以区分选择的两种不同“模式”。这样就得到了以下版本：

$$
\begin{array}{l}
g = 0; f = 0; \\
*[[g \wedge \neg C1.r_a \wedge \neg C1.r_e]; C1.a\downarrow; g\downarrow] \\
\| \\
*[\\
[| \; \neg f \wedge \neg g \wedge C1.r_e \longrightarrow [C1.r_a]; C1.a\uparrow; f\uparrow \\
\; [] \qquad \neg f \wedge C2.r \longrightarrow C2.a\uparrow; [\neg g \wedge \neg C2.r]; C2.a\downarrow \\
\; [] \qquad f \wedge C2.r \longrightarrow g\uparrow; [\neg g]; f\downarrow \\
\; [] \qquad f \wedge \neg C1.r_e \longrightarrow g\uparrow; f\downarrow \\
|] \\
]
\end{array}
$$

最后，考虑到四路仲裁实际上有两组各含两个互不相交的守卫条件，且这两组守卫条件相当相似，我们尝试将其分解为单个仲裁器。如果仲裁器的“模式”由 f 决定，那么仲裁器的一个守卫条件就很简单，就是 $C2.r$。对于另一个守卫条件，请注意，如果第四个守卫条件的合取中包含一个 $\neg g$ 项，那么第一个和第四个守卫条件就可以合并。观察到进入仲裁器的“f-模式”时， g 始终为低，我们可以看出，向这个守卫条件添加一个 $\neg g$ 项是有效的强化。这样做之后，单个仲裁器的第二个守卫条件就会简化为 $G = \neg g \wedge (f \oplus C1.r_e)$。

因此，我们可以将进程中的仲裁分解为一个在两种不同模式下运行的简单仲裁器，如下所示。这就是我们转换为生产规则以实现该电路的最终形式。

$$
\begin{array}{l}
g = 0; f = 0; \\
*[[g \wedge \neg C1.r_a \wedge \neg C1.r_e]; C1.a\downarrow; g\downarrow] \\
\| \\
*[[| \quad G \longrightarrow v\uparrow; [\neg G]; v\downarrow \\
\quad [] C2.r \longrightarrow u\uparrow; [\neg C2.r]; u\downarrow \\
\; |]] \\
\| \\
*[[\; \neg f \wedge v \longrightarrow [C1.r_a]; C1.a\uparrow; f\uparrow \\
\quad [] \; \neg f \wedge u \longrightarrow C2.a\uparrow; [\neg g \wedge \neg C2.r]; C2.a\downarrow \\
\quad [] \; f \wedge u \longrightarrow g\uparrow; [\neg g]; f\downarrow \\
\quad [] \; f \wedge v \longrightarrow g\uparrow; f\downarrow \\
\;]]
\end{array}
$$

不难验证，现在的四路选择是确定性的，因为 u 和 v 是仲裁器输出，并且任意时刻它们中只有一个可以为高电平。

VI. 对称情形

前文已简要介绍对称情形：当两个之字形的权重均为正时，就会出现这种情况。也就是说，C1 提前释放与实际释放之间可能出现 C2 的请求，而 C2 提前释放与实际释放之间也可能出现 C1 的请求，并且在这两种情况下都能保证互斥。由此得到的电路具有以下握手展开形式，可很容易地从非对称情况推导得出。现在存在的 6 路非确定性选择，是对之前 4 路选择的扩展；额外的两种情况是“f-模式”的对称对应情况。

$$
\begin{array}{l}
g1 = 0; g2 = 0; f1 = 0; f2 = 0; \\
*[[g1 \wedge \neg C1.r_a \wedge \neg C1.r_e]; C1.a\downarrow; g1\downarrow] \\
\| \\
*[[g2 \wedge \neg C2.r_a \wedge \neg C2.r_e]; C2.a\downarrow; g2\downarrow] \\
\| \\
*[\\
[| \; \neg f1 \wedge \neg f2 \wedge \neg g1 \wedge C1.r_e \longrightarrow [C1.r_a]; C1.a\uparrow; f1\uparrow \\
\; [] \; \neg f1 \wedge \neg f2 \wedge \neg g1 \wedge C2.r_e \longrightarrow [C2.r_a]; C2.a\uparrow; f2\uparrow \\
\; [] \quad f1 \wedge \neg f2 \wedge \neg g2 \wedge C2.r \longrightarrow g1\uparrow; [\neg g1]; f1\downarrow \\
\; [] \qquad f1 \wedge \neg f2 \wedge \neg C1.r_e \longrightarrow g1\uparrow; f1\downarrow \\
\; [] \quad \neg f1 \wedge f2 \wedge \neg g1 \wedge C1.r \longrightarrow g2\uparrow; [\neg g2]; f2\downarrow \\
\; [] \qquad \neg f1 \wedge f2 \wedge \neg C2.r_e \longrightarrow g2\uparrow; f2\downarrow \\
|] \\
]
\end{array}
$$

与之前一样，这可以分解为一个在 3 种模式下运行的仲裁器，模式由 $f1$和 $f2$决定。在第三和第五个守卫条件中，我们使用简写 $Cx.r = Cx.r_a \wedge Cx.r_e$。

VII. 电路设计与评估

我们在 TSMC 65nm 工艺节点上实现了上述最终电路，并使用 Xyce[11] 进行了仿真。所有仿真均在原理图级别完成。为求完整，我们也列出了能耗指标。非对称情况的设计在两个信道上各进行一次握手、共两次握手的序列中，能耗为 133fJ，总漏电功耗为 876nW。我们使用脉冲源产生请求，测量了 C1 和 C2 端口的响应延迟。在此测试条件下，非对称电路的 C2 端口延迟为 330ps，C1 端口延迟为 910ps。差异源于电路和信道模型的不对称性。C1 的转换序列比 C2 更长，因为还需要额外计算，以便配置电路来响应可能提前到达的 C2 请求。

对称情形下，每次握手序列对应的能量和电路的漏电功率分别为 201fJ 和 1.61μW。两个端口的延迟均为 1.05ns，正如预期的那样，由于电路更加复杂，延迟略长。

SPICE 仿真结果如图 6 和图 7 所示。仅展示了相关的输入变量和输出变量，其中

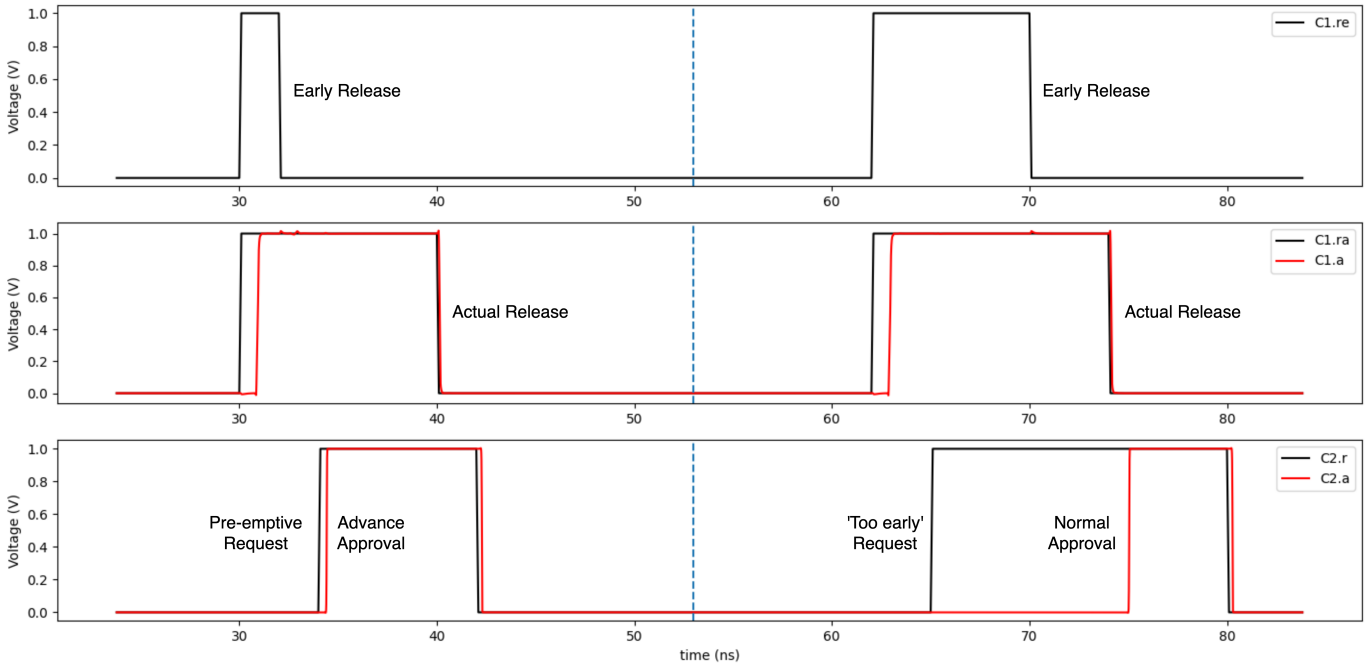


图 6. 非对称情况下电路的 SPICE 仿真波形。由环境生成的请求以黑色显示。由电路生成的确认以红色显示。虚线左侧的情形中，C1 上的两个请求均被置位，随后收到相应的确认。然后，C1 的早期请求被撤销（上方），而 C1 的实际请求（中间）尚未撤销时，C2 请求（下方）便被置位，从而实现了上文所述的提前批准。虚线右侧的情形中，C1 上的两个请求均被置位，随后收到相应的确认。然后，在 C1 的早期请求尚未撤销之前，C2 请求便被置位。此时，服务器必须等待 C1 完全完成后才能确认 C2。在一组握手序列（30 ns–45 ns）中消耗了 133 fJ。

电路的输入（请求）以黑色表示，电路生成的输出（确认）以红色表示。两个确认信号重叠的时间段即为利用时序之字形所获得的延迟缩减。

尽管在非对称情况下会增加约 600ps 的延迟，但使用该电路时，我们预期能够利用之字形的有趣情况会频繁出现。之字形的权重 $W_2 - W_1$ 实际上没有上限，因为它由 C1 和 C2 进程决定，这两个进程会向服务器提供它们使用共享资源所需时间的信息。该时间可能比此电路的延迟大几个数量级，从而显著降低延迟。将对称电路与 330ps 的基准延迟相比，可知会额外产生 ~700ps 的延迟。

这些延迟可用于判断，机会互斥机制的开销是否能被利用时序之字形所获得的收益抵消，无论采用对称情形还是非对称情况。

VIII. 讨论

本文所述的电路比用于处理两个客户端之间互斥的简单仲裁器复杂得多。然而，在已知某些时序信息的情况下，额外的硬件成本值得，因为它能减少关键资源的空闲时间。

最终将其简化为仅使用一个仲裁器的形式，存在一个在使用三个/两个仲裁器的情况下不会出现的缺点。将四路非确定性选择分解为单个仲裁器并非没有代价。当使用保护条件在两次使用之间发生变化的仲裁器时，切换过程中可能会出现不稳定性。为说明这一点，请考虑以下情况：仲裁器中的模式切换（保护条件的改变）必须由某个控制变量上的转换触发，在本例中，该变量为 f。由于它出现在多个进程中，f 现在成为一个共享变量。在传统的共享变量使用方式中，两个进程之间会进行通信，以确保它们不会同时尝试修改/读取该变量。这样可以确保进程访问该变量时，其值是正确的。然而，在我们的电路中，一旦 f 被置位，仲裁器中就不可能发生任何转换并将信号发送回主进程，以确认保护条件的改变已经完成。仲裁器只有三个可能的输出变量：u、v 和 G。但在 f 转换结束时，这些变量都没有唯一确定的值，因为它们都直接取决于外部变量的值。因此，主

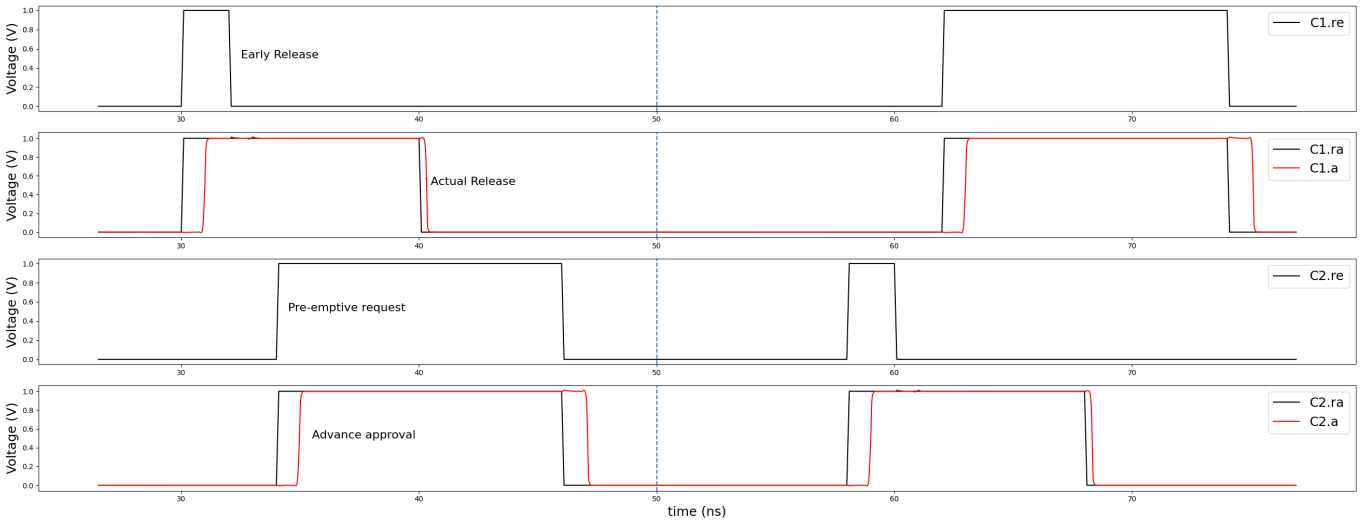


图 7. 对称情形下电路的 SPICE 仿真波形。由环境生成的请求以黑色显示；由电路生成的确认以红色显示。虚线左侧是前文所述的非对称电路情形，其中 C2 接收机会式授权。右侧是其对称对应情形，其中 C1 接收机会式授权。在一组握手序列（30ns - 45ns）中消耗了 201fJ 的能量。

进程 无法 确定保护条件的更改是否已成功完成，除非知道仲裁器中逻辑门的延迟。

如上文所述，当“过载”仲裁器以连续方式执行多次仲裁时，这种不稳定性实际上是根本性的。尽管在所有逻辑门延迟均随机化的仿真场景中这是一个问题，但在实践中，可以通过一个基本的时序假设来解决：要求实现 G 布尔函数的逻辑门，其速度快于从 $f\uparrow$ 到以下任一 $f\downarrow$ 转换的转换序列——这两种转换都包括机会互斥电路中握手变量的变化，随后是环境的响应。通过使用额外的状态变量，该假设可以简化为要求单个反相器的速度快于几个更复杂的逻辑门，而这在实践中很容易实现。我们在第七节中的评估对应于一种将这些额外状态变量作为所报告开销一部分的电路。

最后，由于我们已经在利用时序约束来决定是否使用该电路，因此这一额外的局部约束可以通过设计来保证，不会造成显著限制。

第九节：结论

在本文中，我们提出了一种新颖的方法，用来利用某些细微的时序约束来设计互斥服务器。我们展示了，这些服务器可以基于对情况的了解，通过机会性地同时允许多个客户端访问，来减少共享资源的空闲时间。关于客户端停止和开始使用资源的预期时间。我们介绍了进程的握手扩展，在台积电 65 纳米工艺节点中实现了相同的设计，并评估了其性能。最后，我们讨论了实际适合使用该电路的场景，以及它的优点和潜在缺点。所有这些都展现了之字形因果关系这一概念的价值；该概念最初由 [9], 在电路设计中提出。据我们所知，这是首项如此应用该概念的工作。

致谢

本研究部分得到了 DARPA IDEA 项目资助 FA8650-18-2-7850 和 DARPA POSH 项目资助 HR001117S0054-FP-042 的支持。Yoram Moses 是以色列理工学院的 Israel Pollak 学术讲席教授，并部分获得了 BSF 项目资助 2015820 的支持，该资助与 NSF-BSF 项目资助 CCF 1617945 同时获得。

参考文献

[1] L. Lamport，“一种快速互斥算法”，*ACM* 计算机系统汇刊（*TOCS*），第5卷，第1期，第1–11页，1987年。[2] M. Raynal和D. Beeson，互斥算法。麻省理工学院出版社，1986年。[3] E. W. Dijkstra，“并发程序控制问题的解”，*ACM*通讯，第8卷，第569页，1965年9月。[4] A. J. Martin，“进程环上的分布式互斥”，《计算机编程科学》，第5卷，第265–276页，1985年。[5] A. J. Martin，“用于令牌环互斥的异步电路”，1990年。[6] K. Raymond，“一种用于分布式互斥的基于树的算法”，*ACM* 计算机系统汇刊（*TOCS*），第7卷，第1期，第61–77页，1989年。[7] W. Hua和R. Manohar，“异步系统的精确时序分析”，*IEEE* 集成电路与系统计算机辅助设计汇刊，第37卷，第1期，第203–216页，2017年。

[8] W. Hua、Y.-S. Lu、K. Pingali和R. Manohar，“Cyclone：用于异步电路的静态时序与功耗引擎”，载于*2020*年第*26*届*IEEE*异步电路与系统国际研讨会（*ASYNC*），第11–19页，IEEE，2020年。[9] A. Dan、R. Manohar和Y. Moses，“通过之字形因果关系实现无时钟时间的使用”，载于*ACM*分布式计算原理研讨会论文集，第241–250页，2017年。[10] R. Manohar和Y. Moses，“定时信号进程”，载于*IEEE*异步电路与系统国际研讨会，IEEE，2023年。[11] E. Keiter、T. Russo、R. Schiek、H. Thornquist、T. Mei、J. Verley、K. Aadithya和J. Schickling，“Xyce并行电子仿真器用户指南7.6版”，技术报告，桑迪亚国家实验室（SNL-NM），美国新墨西哥州阿尔伯克基，2022年。[12] C. A. R. Hoare，“通信顺序进程”，*ACM*通讯，第21卷，第8期，第666–677页，1978年。[13] A. J. Martin，“异步VLSI电路的综合”，1991年。[14] N. Bingham和R. Manohar，“仲裁表达式的系统化方法”，《*IEEE*电路与系统汇刊*I*：常规论文》，第67卷，第12期，第4960–4969页，2020年。[15] R. Manohar，“重排握手展开分析”，载于第七届异步电路与系统国际研讨会论文集。*ASYNC 2001*，第96–105页，IEEE，2001年。

附录I

通信硬件进程（CHP）是一种硬件描述语言，用于描述源自 C.A.R. Hoare 的通信顺序进程（CSP）的无时钟电路 [12]。 [13] 中提供了 CHP 及其语义的完整描述。下面非正式地介绍我们所使用的该符号表示法的一个子集，并按优先级从高到低列出，内容转自 [14]。有关发送和接收等通道动作的握手展开与组合运算符之间交互的完整讨论，请参阅 [15]。

一个**信道** X由**请求** X.r以及**确认** X.a或**使能** X.e组成。确认和使能的作用相同，但含义相反。信道通过这些信号实现网络协议，将数据从一个QDI进程传输到另一个QDI进程。

- **跳过**：跳过 不执行任何操作，并继续执行下一条命令。
- **无数据赋值**：$n\uparrow$ 将节点n 设为$true$ ，而$n\downarrow$ 将其设为$false$。
- **探测**：X? 用于确定信道是否已准备好执行接收操作，返回等待请求的值 $X.r$，而不执行接收操作。X! 用于确定信道是否已准备好执行发送操作，根据确认或使能分别展开为 $\neg X.a$ 或 $X.e$ 。对于无数据的信道，语法简化为 X。
- **顺序组合**：S; T执行程序S，随后执行T。
- **并行组合**：$S \,||\, T$ 以任意顺序执行程序S和T 。
- **确定性选择**：$[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]$其中 G_i（称为守卫）是一个无数据表达式，而 S_i是一个程序。选择操作会一直等待，直到其中一个守卫G_i的计算结果为真，然后执行相应的程序S_i。这些守卫必须保持稳定且相互排斥。排斥性的 . 符号表示法 $[G]$ 是 $[G \rightarrow$跳过$]$的简写。，对应于等待G 变为真。
- **非确定性选择**：$[|G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n |]$ 与确定性选择相同，只是守卫不必稳定或互斥。如果两个或更多守卫同时求值为真 ，则会任意选择其中一个（不一定是随机选择）。在电路中，这种选择由一组仲裁器和同步器实现。当两个或更多守卫同时求值为真 时，仲裁器或同步器可能进入亚稳态。随后该亚稳态以非确定性的方式消解，将授权授予选择语句的某一个分支。因此，在这种情况下，该选择语句的数字模型同样具有非确定性。
- **重复**：$*[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]$ 与选择语句类似。不过，动作会重复执行，直到所有守卫条件的计算结果都不再是真。$*[S]$ 是[真 $\rightarrow S$]的简写。

附录II

服务器进程（非对称情况）的可由 CMOS 实现的生产规则集如下所示。一对生产规则共同定义了实现该变量的组合逻辑/状态保持逻辑门。带下划线前缀的变量通常用于表示变量的反相形式。组合逻辑门以 **(*)** 标记，因此省略了该变量对应的另一条生产规则。 u, v 的生产规则使用仲裁器实现。

$$\neg Reset \rightarrow _Reset\uparrow\ (*)$$
$$\neg _Reset \vee \neg C1.r_e \rightarrow _C1.r_e\uparrow\ (*)$$

$$\neg u_{reg1} \rightarrow u_{reg1}\uparrow (*) \neg u$$
$$_{reg2} \rightarrow u_{reg2}\uparrow (*) \neg v$$
$$_{reg1} \rightarrow v_{reg1}\uparrow (*) \neg v$$
$$_{reg2} \rightarrow v_{reg2}\uparrow (*)$$

$$_u_{reg1} \wedge _u_{reg2} \wedge _v_{reg1} \wedge _v_{reg2} \rightarrow reg\downarrow\ (*)$$

$$\neg reg \wedge \neg g \wedge$$
$$((\neg _f \wedge \neg C1.r_e) \vee (\neg f \wedge \neg _C1.r_e)) \rightarrow G_{arb}\uparrow$$
$$Reset \vee reg \wedge (v_{reg1} \vee v_{reg2}) \rightarrow G_{arb}\downarrow$$

<escape>$u, v = Arbiter(G_{arb}, C2.r)$
</escape>

$$\neg u \rightarrow _u\uparrow\ (*)$$
$$\neg v \rightarrow _v\uparrow\ (*)$$

$$(\neg f \wedge \neg C1.a \wedge \neg g) \rightarrow _g_{reg}\uparrow$$
$$C1.a \wedge v_{reg1} \wedge reg \rightarrow _g_{reg}\downarrow$$
$$\neg _g_{re} \rightarrow g_{reg}\uparrow\ (*)$$

$$\neg u \wedge \neg _f \wedge \neg reg \rightarrow u_{reg1}\uparrow$$
重置 ∨(寄存器 $\wedge _g \wedge f \wedge _C1.a) \rightarrow u_{reg1}\downarrow$

$\neg v \wedge \neg f \wedge \neg reg \rightarrow v_{reg1}$↑重置
$\vee (reg \wedge f \wedge v) \rightarrow v_{reg1}$↓$\neg u$
$\wedge \neg f \wedge \neg reg \rightarrow u_{reg2}$↑重置 $\vee$
$(\bar{reg} \wedge g \wedge u) \rightarrow u_{reg2}$↓

$\neg v \wedge \neg f \wedge \neg reg \wedge \neg g_{reg} \rightarrow v_{reg2}$↑
重置 $\vee (reg \wedge C1.a \wedge f \wedge v) \rightarrow v_{reg2}$↓

$\neg$重置 $\vee (\neg C1.a \wedge \neg f) \rightarrow g$↑$((reg \wedge$
$u^{reg1}) \vee g^{reg}) \wedge f \rightarrow g$↓$\neg g \rightarrow g$↑(*)

$\neg$重置 $\vee((\neg u_{reg1} \wedge \neg C1.a) \vee \neg v^{reg1}) \wedge \neg$
$g \rightarrow f$ ↑$reg \wedge v_{reg2} \wedge C1.a \rightarrow f$ ↓$\neg f \rightarrow f$↑
重置 $\vee (g \wedge f) \rightarrow f$↓

$\neg$重置 $\vee(\neg v_{reg2} \wedge (\neg u_{reg1} \vee \neg g^{reg}) \wedge \neg C1.r_a \wedge \neg$
$C1.r_e) \rightarrow C1.a$↑$g_{reg} \wedge C1.r_a \wedge C1.r_e \wedge v_{reg2} \wedge$
$reg \rightarrow C1.a$↓$\neg C1.r_e \wedge \neg C1.a \rightarrow C1.a$↑重置 $\vee ($
$C1.r_e \wedge C1.\bar{a}) \rightarrow C1.a$↓

$\neg C2.r \wedge \neg u_{reg2} \rightarrow C2.a$↑$reg \wedge$
$u_{reg2} \wedge C2.r \rightarrow C2.a$↓$\neg C2.a \rightarrow C2.a$↑
(*)