arXiv:2304.04488v1 [cs.DC] 10 Apr 2023

面向交互式数据中心应用的混合计算

普拉蒂尤什 Patel
pratyush@cs.washington.edu
华盛顿大学
USA

凯蒂·林
katielim@cs.washington.edu
华盛顿大学
USA

库沙尔·琼琼瓦拉
kushaljh@cs.washington.edu
华盛顿大学
USA

Ashlie Martinez
ashmrtnz@cs.washington.edu
华盛顿大学
USA

Max Demoulin
maxdml@seas.upenn.edu
天文学家
USA

Jacob Nelson
jacob.nelson@microsoft.com
微软研究院
USA

Irene Zhang
irene.zhang@microsoft.com
微软研究院
USA

托马斯·安德森
tom@cs.washington.edu
华盛顿大学
USA

摘要

对于各种数据中心应用程序而言，现场可编程门阵列（FPGA）的能效和成本效益均优于 CPU。然而，对于延迟敏感且具有突发性的工作负载，由于 FPGA 的启动成本高昂，这一优势可能难以发挥。我们提出一种 FPGA 与 CPU 混合计算框架，可让此类工作负载以合理的成本利用 FPGA 的能效优势。我们的核心见解是，使用 FPGA 处理稳定状态下的工作负载，并使用 CPU 处理短期工作负载突发。基于这一思路，我们设计了轻量级混合调度器 Spork，使这些能效与成本优势能够在实际中得以实现。根据目标不同，Spork 可以在能效与降低成本之间进行权衡，反之亦然。该调度器通过参数设定 FPGA 与 CPU 在功耗、性能、成本和启动延迟方面的关键差异。我们改变这些参数，并基于生产环境与合成跟踪数据分析各种应用与工作节点配置。对云工作负载的评估表明，能效优化型 Spork 不仅能效更高，也比同构平台更便宜——对于截止时间紧迫、请求较短的应用程序，其能效提高 1.53× 倍，成本也比仅使用 FPGA 低 2.14× 倍。与现有成本优化型混合调度器的理想化版本相比，能效优化型 Spork 在成本相当的情况下能效提高 1.2–2.4× 倍；而成本优化型 Spork 的能效提高 1.1–2× 倍，成本降低 1.06–1.2× 倍。

关键词

能效、数据中心、调度、混合计算、无服务器计算、加速器、现场可编程门阵列、CPU、建模与分析

1 引言

云应用程序不断增长的能量需求，使数据中心运营商及其客户重新关注能效。对于许多重要的数据中心应用程序而言，现场可编程门阵列的能耗更低，且单位成本性能优于中央处理器 [8, 10, 15, 39, 59, 78]。然而，将延迟敏感型和突发型应用卸载到现场可编程门阵列共享池中，因其较高的启动成本而变得复杂。要启动新的处理实例以响应应用程序请求突发，必须对现场可编程门阵列进行重新配置；这一过程缓慢且能耗高，可能需要数秒 [10, 20, 38]。此类启动延迟往往会导致无法满足请求截止时间。反复产生的启动成本也可能抵消能效优势。因此，必须为现场可编程门阵列配置过量资源，以应对延迟敏感型和突发型流量 [23, 65]。

本文旨在展示如何利用 FPGA 和 CPU 混合平台，即使在成本合理的情况下，也能为延迟敏感型和突发型应用发挥现场可编程门阵列的能效优势。我们观察到，现场可编程门阵列和 CPU 各具互补优势：现场可编程门阵列通常消耗更少能量，并且在稳定工作负载下表现更好；而 CPU 能够适应工作负载变化，并且在低负载下成本更低。能够交替运行应用程序请求的混合平台，可能比单独使用现场可编程门阵列或 CPU 更高效且成本更低。

混合平台面临两项关键挑战。首先，程序员必须为 FPGA 工作节点和 CPU 工作节点分别开发和维护应用程序实现 [13, 36, 84, 94]。其次，编排框架需要经过修改，才能跨异构工作节点运行请求 [20, 35, 68, 69, 80]。尽管此前已有一些工作尝试解决这些问题，但要在实践中实现 FPGA-CPU混合计算，仍需付出额外努力。因此，我们分析混合平台所带来的能效和成本效益是否足以证明这些努力是合理的。

我们提出了一种 FPGA-CPU 混合计算框架，使用现场可编程门阵列处理稳定状态下的工作负载，并使用中央处理器处理频率高于 FPGA 启动时间所能应对的请求突发。在关于调度器、工作负载和计算工作节点的某些假设下，我们证明，对于突发性中等至较高的应用程序，该框架能够比纯 FPGA 平台实现更高的能效，同时成本也低得多。通过适当配置，混合平台还可以根据需要在这些能效提升与降低成本之间进行权衡。

我们设计了 Spork，这是一款轻量级混合计算调度器，在现场可编程门阵列和中央处理器工作节点供应充足的情况下，可在实践中实现这些优势。Spork 会根据服务负载所需工作节点数量的历史分布，在每个调度区间主动分配现场可编程门阵列工作节点。分配方案旨在使预期的加权和最小化】【。

后续时间间隔内的能耗与成本。随后，Spork 会在该时间间隔内调整分配，以确保满足请求截止时间，并假设请求大小已预先知晓。具体而言，Spork 会通过在请求分发路径上动态分配短时运行的 CPU 工作线程，来处理工作负载突发以及 FPGA 分配不足（由误预测导致的情况）。最后，Spork 会派发请求，以最大限度地提高现场可编程门阵列利用率，并快速回收空闲工作节点，从而降低能耗和成本。

由于不同的现场可编程门阵列和中央处理器工作节点具有不同的特性，且这些特性也会受到应用程序行为的影响，因此我们根据工作线程功耗、性能、成本和启动开销方面的关键差异对 Spork 进行参数化。通过改变这一参数空间，我们首次在大量工作节点和工作负载配置下，基于生产环境和合成跟踪数据，对混合平台与同构平台之间的能效和成本权衡进行了详细研究。

我们对云工作负载的评估表明，在对截止时间以及 FPGA 与 CPU 的性能和能耗比做出合理假设的情况下，Spork 的能效显著高于同构平台，且成本更低。对于截止时间紧迫的生产环境跟踪数据，能耗优化型 Spork 的能效高出 1。53×，成本低 2。14×。与理想化的 MArk 相比，[93], 这是一种最先进的成本优化型混合调度器，能耗优化型 Spork 在成本相近的情况下能效高出 1.2–2。4×，而成本优化型 Spork 的能效高出 1.1–2×，成本低 1.06–2×。与没有启动和空闲开销的理想化最佳情况纯 FPGA 系统相比，能耗优化型 Spork 仅多消耗 7–14% 的能量，成本优化型 Spork 的成本仅高出 5–17%。

我们的贡献如下：

- 我们通过展示在理想化设置下，混合平台能够以更低的成本，为对延迟敏感且具有突发性的工作负载利用现场可编程门阵列的能效优势，从而阐明 FPGA-CPU 混合计算的动机（§ 3）。
- 通过配置优化目标，我们表明混合平台能够逐步权衡这些能效提升，以换取更大幅度的成本降低（§3）。
- 我们设计了 Spork，这是一种实用的混合调度器，借助轻量级预测，在实践中实现了这些优势，并
  高效的请求派发（§ 4）。
- 我们在生产环境和合成跟踪数据上评估了 Spork 的三种变体，并表明在多种合理的工作负载和工作节点配置下，它们的性能显著优于最先进的同构平台和混合平台（§ 5）。

2 背景

2.1 数据中心平台

图 1 展示了 Spork 的高层架构，其设计与无服务器框架类似 [3, 20, 41, 62]。我们的目标是让延迟敏感型、突发性无状态函数在由现场可编程门阵列和中央处理器工作节点组成的共享池上运行 [24, 28, 42, 75, 86]。客户端通过向调度器发送请求来调用应用程序，调度器执行两项关键操作：(1) 从池中分配（启动）和释放（关闭）工作节点，以处理到达负载；(2) 将请求派发给已分配的工作节点，并将结果返回给客户端。应用程序请求必须遵守预先设定的延迟期限——

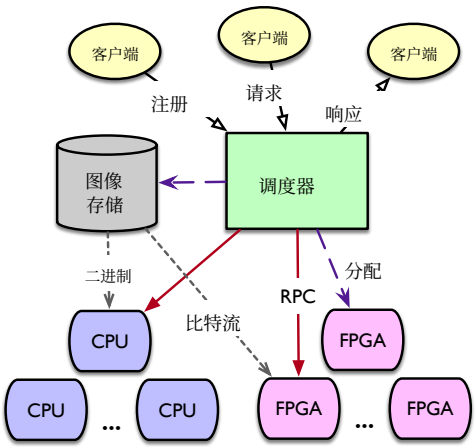


图 1：Spork架构。

	CPU		FPGA	
部署	启动		重新配置	快速启动
容器 [53]	~500毫秒		闪存	~1–3分钟
微型虚拟机 [2]	~125毫秒		JTAG [38]	~10–30秒
进程 [21]	~5ms		PCIe [73]	~1–10秒
协程 [19]	<1ms		内部配置访问端口* [47, 54]	~500毫秒

表 1：不同中央处理器和现场可编程门阵列部署的启动延迟。*内部配置访问端口是一种部分重配置技术，无法用于任意应用逻辑。

应用程序	能效	加速比
卷积神经网络推理 [76]	2.1–5.3×	1–7.7×
RNN推理 [58]	1.8–4.5×	0.75–4×
网络搜索 [65]	N/A	1.95×
图像处理 [66]	5.68×	N/A
入侵检测 [96]	1.4–3×	5.8–8.8×
文档过滤 [12]	6.19×	0.85×
视频压缩 [11]	5.2×	8.4–10.1×
Bzip2 压缩 [67]	N/A	1.6–2.3×
高性能计算内核 [57]	2–6×	0.07–1×
键值存储 [48]	2.18×	1.02×
数据库 [39]	1.42×	1–2×
几何平均值	3.32×	1.88×

表 2：各种应用程序中现场可编程门阵列的繁忙功耗效率及相对于CPU 的处理加速比。数值源自所引用的研究；大多数研究比较的是单插槽 CPU 与性能等效的现场可编程门阵列。

例如，客户端可能要求请求在 10× 请求服务时间 [55]内完成。

2.2 现场可编程门阵列与中央处理器的权衡

现场可编程门阵列和CPU 在多个方面存在差异，这些差异可能会影响平台的能效和成本。我们将在下文对此进行讨论。

启动延迟。 FPGA工作线程的启动时间比CPU长，且消耗更多能量 [18, 20, 21, 38]。实际上，启动延迟取决于部署模式，如表 1所示。重新配置是现场可编程门阵列的主要延迟瓶颈。频繁启动现场可编程门阵列也会增加能耗，因为重新配置

耗电量大，而且无法并行执行有用工作。[1] 部分重配置速度更快，可在一秒内完成 54]

[47,

。

然而，由于现场可编程门阵列设计和布局规划方面的限制，它无法用于任意应用逻辑 [37]，，因此本文不予考虑。这些因素导致现场可编程门阵列在大多数数据中心应用程序中采用静态配置 [65, 73]，，而中央处理器通常会动态配置以匹配负载 [18, 25, 41, 75]。

效率与性能。 现场可编程门阵列运行相同的应用程序时，通常比中央处理器消耗更少的能量。能量取决于功率和执行时长。得益于专门设计的实现，现场可编程门阵列在这两方面往往表现更佳。专门化消除了中央处理器中低效的指令获取和解码流水线，并使现场可编程门阵列能够充分利用最适合各应用程序的并行性。表 2 列举了多项研究，展示了将任务卸载到现场可编程门阵列带来的优势。请注意，这些研究采用了不同代际的硬件。对于这些应用程序和硬件配置，现场可编程门阵列的能效平均比同等的单路中央处理器高 ~6。25× (3.32 × 1.88)。即使某个应用程序在现场可编程门阵列上的性能较低（例如文档过滤），总体而言其能效往往仍然更高。

空闲功耗。 即使处于空闲状态，已启动的工作节点也会消耗功率。CPU 的能量比例性（空闲与活跃状态下功率消耗之比）通常要好得多。例如，如今空闲状态下的 CPU 仅消耗其在最大利用率 [7] 时功率的 ~15%。大多数现场可编程门阵列本身并不支持动态电压与频率调节（DVFS）和休眠状态等降低功率的技术，不过可以采用特定设计来减少空闲功耗浪费 [33, 95]。然而，由于 CPU 的最大功率更高，这两个平台的空闲功耗往往相近。

成本。 现场可编程门阵列通常比可比-较的中央处理器产生更高的资本支出，并且每个工作节点实例的成本也更高。例-如，在 AWS 云 [74] 上：一台 8v中央处理器虚拟机（x2gd.2xlarge）的成本为 67¢/小时，而一台 F1 现场可编程门阵列（ f1.2xlarge 减去等效的中央处理器主机虚拟机）的成本为98¢/小时（~1.5× 差异）。类似趋势也适用于阿里云的 F3 现场可编程门阵列实例 [16, 17]。

3 支持混合计算的理由

我们通过在一种理想化场景下，评估混合平台和同构平台上的延迟敏感型工作负载与突发型工作负载，从而深入理解混合计算；在该场景中，任务到达情况是预先已知的。在下一节中，我们将利用这些洞见，在不作此假设的情况下，开发一种实用且高效的混合调度器。

3.1 帕累托最优调度器

表3 展示了能量最优混合调度器的参数。该调度器使用完美的工作负载信息，将应用程序的总体能耗降至最低。调度器通过求解混合整数线性规划（MILP），确定随时间分配的工作节点数量和类型。混合整数线性规划约束确保：（1）在每个调度区间内分配足够的工作节点，以满足该区间内到达的所有请求；（2）根据未来负载信息，在释放工作节点和使其保持空闲之间作出最高效的决策。我们对

[1]在新的应用程序比特流写入闪存时，可以使用现场可编程门阵列上的现有逻辑。

	输入
$X_t \in [0,\infty)$	在时间 $[t, t+1$ 到达的请求数量)
a^c, a^f	中央处理器、现场可编程门阵列的分配能耗
d^c, d^f	中央处理器、现场可编程门阵列的释放能耗
e_b^c, e_b^f	一个时间间隔内中央处理器、现场可编程门阵列的忙碌功率
e_i^c, e_i^f	时间间隔内中央处理器、现场可编程门阵列的空闲功耗
r^c, r^f	中央处理器、现场可编程门阵列的请求处理速率
S	现场可编程门阵列的启动延迟
	输出
$Y_t^c \in [0, N_c]$	随时间分配的中央处理器总数 $[t, t+1)$
$Y_t^f \in [0, N_f]$	随时间分配的现场可编程门阵列总数 $[t, t+1)$
$B_t^c, B_t^f \in [0, N_r]$	繁忙的 CPU、现场可编程门阵列随时间变化 $[t, t+1)$
	辅助项
$I_t^c, I_t^f \in [0, N_r]$	空闲的 CPU、现场可编程门阵列随时间变化 $[t, t+1)$
$E_t^a, E_t^d, E_t^b, E_t^i$	$[t, t+1)$ 中分配、释放、繁忙、空闲的总能量
	约束条件
min	$\sum_t (E_t^a + E_t^b + E_t^i + E_t^d)$
s.t.	$\forall t: E_t^a = \sum_{w\in\{c,f\}} \max(Y_{t+1}^w - Y_t^w, 0)\, a^w$
	$\forall t: E_t^b = \sum_{w\in\{c,f\}} e_b^w B_t^w$
	$\forall t: E_t^i = \sum_{w\in\{c,\ f\}} e_i^w I_t^w$
	$\forall t: E_t^d = \sum_{w\in\{c,f\}} \max(Y_t^w - Y_{t+1}^w, 0)\, d^w$
	$\forall t: r^c B_t^c + r^f B_t^f = X_t$
	$\forall t: Y_t^c = B_t^c + I_t^c \le N_c$
	$\forall t: Y_t^f = B_t^f + I_t^f \le N_f$
	$\forall t: Y_{t+S}^f \ge \sum_{\tau=t}^{t+S} \max(Y_{\tau+1}^f - Y_\tau^f, 0)$

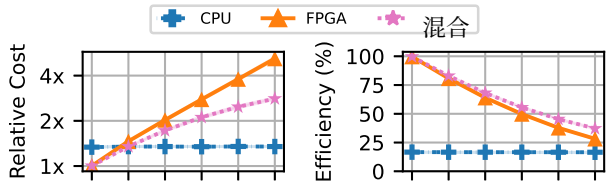
表3：在工作负载信息完美的情况下，每个时期的混合整数线性规划（MILP）调度器在中央处理器（c）和现场可编程门阵列（f）工作节点之间进行调度，以最小化能耗。成本最优公式与此类似；其他帕累托最优公式则最小化能量和成本的加权和。

同构平台采用相同的公式，但会限制可分配的工作节点类型。我们还为现场可编程门阵列增加了一项约束，确保其分配时长至少达到启动时长。成本最优调度器采用类似的公式；它不区分空闲和繁忙的工作节点，而是仅根据工作节点的启动时长估算成本。其他帕累托最优调度器则最小化能量和成本的加权和。

除了拥有完美的未来工作负载信息之外，这些调度器还做出了两个不切实际的假设。首先，它们假设工作节点分配和取消分配都是瞬时完成的；但实际上，这些操作仍会产生能量和成本开销。其次，它们假设所有到达的请求也都能在同一时间间隔内完成，不会跨时间间隔溢出。这些假设结合起来，简化了约束公式，并有助于粗略估算混合计算相较于同构平台的效率和成本效益。具体而言，它们针对现场可编程门阵列进行了优化——在实践中，误预测很可能发生，从而导致纯 FPGA 平台需要过度配置、延迟容忍，或两者兼具。

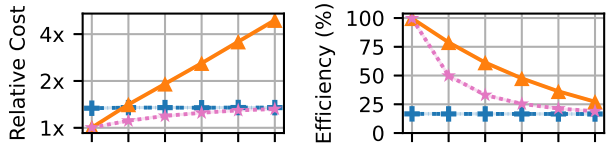
3.2 混合与同质平台

假设。 根据我们在 § 2.2 中的调查，我们假设 FPGA 工作节点在处理请求时比 6×CPU 的能效更高（CPU 功率为 150W，FPGA 功率为 50W，FPGA 的速度 2× 更快），且空闲时功率更低（CPU 为 30W，FPGA 为 20W [57]）。CPU 工作节点启动耗能 0.75 J（以 150W 功率启动需 5毫秒），而 FPGA 工作节点重新配置耗能 500 J（以 50W 功率启动需 10秒）。工作节点的定价设定为 FPGA 的使用成本比 CPU 高 1.5× 倍（CPU 每小时 67¢，FPGA 每小时 98¢）。请注意，此成本



工作负载突发性

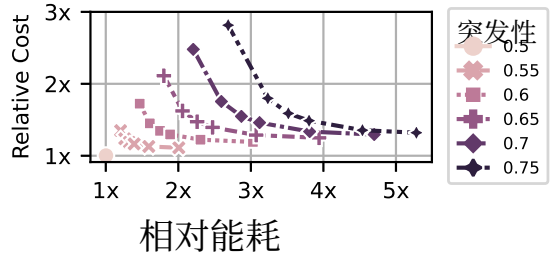
(a) 能耗最优调度。



工作负载突发性

(b) 成本最优调度。

图 2：随着工作负载突发性增加，采用最优基于速率的调度器时，仅 CPU、仅 FPGA 和混合平台的能效与成本。结果以无开销的理想纯 FPGA 平台为基准进行归一化，并取十次轨迹运行的平均值。使用 b 模型生成自相似轨迹 [87],其中 b=0.5 表示均匀负载，b=0.75 表示负载变化很大。



相对能耗

图 3：在不同工作负载突发性下，帕累托最优混合整数线性规划调度器之间的能耗与成本权衡，相对于理想纯 FPGA 平台。每个突发性取值对应的边界标记表示能耗最优和成本最优调度器。

不同于总拥有成本（TCO） [7]，它仅表示固定时长内工作节点的占用成本。

我们评估自相似工作负载轨迹，因为云环境中通常会观察到此类分布 [50, 64, 85, 91]。具体而言，我们使用b-模型 [87], 其中偏置参数捕捉工作负载突发性。直观地说，突发性值为0.5 对应于均匀轨迹，而 0.75则意味着极高的可变性（某些连续时间间隔内的负载差异超过 ~20×）。每条轨迹持续一小时，平均速率为每秒 10,000 个请求。请求大小固定，并在CPU 工作线程上于 10ms 内运行完成。所有请求都在其到达的同一时间间隔内完成。工作节点数量充足，并会扩容或缩容以匹配传入需求。

结果。 图2展示了随着工作负载突发性增加，均质平台和混合平台的能效与成本。结果均相对于理想化的纯FPGA平台，该平台仅产生计算成本（即启动和空闲开销为零）。每个数据点均为十次随机生成的轨迹运行的平均值。

在同构平台中，我们发现，现场可编程门阵列更适合处理突发性较低的工作负载（比 CPU 便宜至多 1.3×，能效高出6×），而 CPU 在突发性较高时表现良好（便宜至多 4×，能效仅比现场可编程门阵列低 1.7×）。相比之下，混合平台兼得两者之长——在优化指标上优于或不逊于同构平台。

在能量最优的设置下，低突发性时，仅 FPGA 和混合方案的能效相近，因为此时最好使用现场可编程门阵列处理大多数请求。然而，在高突发性时，混合方案的能效最高可提升 1.3× 倍，成本最高可降低 1.8× 倍，因为它还可以利用 CPU 处理短暂的突发负载。在低到中等突发性时，混合方案占优——其能效略高于仅 CPU 或仅 FPGA 方案，成本也更低。

成本优化的混合平台比任一同构平台都便宜，同时能效也远高于仅 CPU 平台。与仅使用现场可编程门阵列相比，在工作负载突发性较高时，混合平台的成本最高可降低 3.8× 倍，而能效仅低 1.4×。在中等突发性时，这一优势会减弱：混合方案的成本最高可降低 1.6× 倍，能效则低 1.8×。

无论优化能量还是成本，同构平台的结果都相近。相比之下，混合平台可以通过调整目标权重，在两项指标之间进行权衡，如图 3所示的帕累托最优曲线。这种权衡可能相当显著。例如，在高突发性下，能量最优的混合平台成本比成本最优平台高出 2× 以上。由于现场可编程门阵列的能效远高于 CPU，能量最优的做法是在平均利用率较低时也启动现场可编程门阵列，这会增加额外成本。在成本最优的设置下，只有当工作负载足够繁重、使用现场可编程门阵列具有成本效益时，才会启用它。

4 实用混合 调度

为了让 FPGA-CPU 混合计算切实可行，我们需要：(1) 一个编排框架，用于管理异构工作节点并分配应用程序请求；(2) 一个调度器，用于做出工作节点分配和请求调度决策，以实现理想的能效和成本权衡。先前的工作已开发出框架，用于支持 FPGA 和 CPU 工作节点管理 [20, 35, 82]以及在特定环境中可互换地执行应用程序 [68, 80]。尽管这些工作并非完整的混合编排平台，却为实现 FPGA-CPU 混合计算提供了构建模块。我们的重点是第二部分——设计一个实用的混合调度器，在能效与成本之间进行权衡，并确定混合计算能够带来显著效益的配置。

我们首先考察现有调度器能否应对这一挑战。表 4 总结了现有调度器采用的设计方案。AutoScale [27] 和无服务器 [75] 是同构调度器，能够降低能耗，并在

调度器	混合?	预测?	分配频率	派发
AutoScale [27]	No	No	长期	索引打包
无服务器 [75]	No	No	短期	1:1
INFaaS [70]	Yes	No	长期	基于利用率的
MArk [93]	Yes	成本高昂的	Both	轮询
Spork	Yes	低成本	Both	效率优先

表 4：调度器摘要。

工作节点快速启动与空闲之间取得平衡。AutoScale 还引入了节能的基于索引的派发机制：按照最繁忙优先的顺序，将工作节点填充至阈值，以便更轻松地关闭空闲工作节点。INFaaS [70] 和 MArk [93] 为在中央处理器和 GPU 上提供推理服务，采用了成本优化、考虑截止时间的混合调度器。INFaaS 依赖响应式启发式方法，因此无法充分发挥效率优势。MArk 使用基于 LSTM 的预测，这种方法成本高昂且难以扩展。MArk 还采用低效的轮询派发机制（见 § 5.3）。至关重要的是，这两种调度器都未考虑能效，也未考虑空闲工作节点与繁忙工作节点之间的差异。为弥补这一不足，我们提出 Spork——一种轻量级混合调度器，它对这些工作节点特性进行建模，在满足请求截止时间的同时优化能耗和成本。

4.1 Spork 概述

为便于说明，我们主要从能量优化的角度介绍 Spork，之后再讨论如何扩展以纳入成本优化及其权衡。

Spork 执行两项关键功能：工作节点分配/解除分配以及请求派发。Spork 的底层策略是使用现场可编程门阵列处理稳态请求，并使用中央处理器处理短期突发和低负载情况。它通过预测所需的工作节点数量，并在调度间隔开始时分配这些工作节点（下限受现场可编程门阵列启动延迟限制），尝试针对每个调度间隔合理确定现场可编程门阵列分配规模。Spork 的预测器通过以下信息进行估算：(1) 在近期调度间隔所需工作节点数量这一条件下，所需现场可编程门阵列工作节点数量的历史分布；以及 (2) 在当前已分配工作节点数量这一条件下，工作节点启动开销在其预期生命周期内的摊销值。具体而言，它会在条件分布的所有候选值中，分配使空闲能耗与繁忙能耗之和的期望值最小的工作节点数量。如果需要，Spork 会在请求分发路径上分配短生命周期的中央处理器工作节点，以弥补现场可编程门阵列分配不足或请求突发，这与典型的无服务器编排方式类似 [3, 41, 62]。工作节点在空闲固定时长后会被解除分配，以防止低效且重复的分配 [27, 75]。

对于请求派发，我们的策略是调度请求，使其满足以下条件：(1)满足截止时间要求；(2) 最大化现场可编程门阵列工作节点的利用率；(3) 快速回收空闲工作节点。这些标准有助于满足延迟要求，并降低能耗和成本。为满足截止时间要求，我们的调度器会在做出派发决策前估算请求大小和工作器完成时间。为最大化能效，它会优先将请求派发到现场可编程门阵列，而不是效率较低的 CPU。为快速回收空闲工作节点，它会尽可能将负载集中到最繁忙的工作节点上，从而释放其他工作节点，使其在空闲超时 [27] 后得以回收。

表5列出了本节中使用的符号。

符号	说明
A_w	工作节点 w 的启动延迟
E_w	工作节点 w上的平均请求服务时间
B_w	工作节点 w 的运行时忙碌功耗
I_w	工作节点 w 的空闲功耗
C_w	工作节点 w 的单位时间成本
S	现场可编程门阵列相对于中央处理器的加速比
n_t	区间 t所需的现场可编程门阵列数量
T_s	调度间隔长度
T_b	现场可编程门阵列相对于中央处理器的效率的盈亏平衡服务阈值

表5：符号汇总。下标 $w \in \{c, f\}$ 分别表示中央处理器和现场可编程门阵列工作节点。

4.2 分时段分配

算法 1 描述了 Spork 的逐区间分配器。由于 FPGA 工作线程的启动时间较长，Spork 会定期调整 FPGA 工作节点分配。分配周期（T_s）的下限为 FPGA 启动时间（A_f）。为简单起见，在此描述中我们假设 $T_s = A_f$。在任一时间间隔 t开始时，Spork 使用前一时间间隔所需的 FPGA 工作节点数量， n_{t-1}、工作线程数量的历史分布， $\mathbb{H}$以及工作线程生命周期的历史分布，$\mathbb{L}$来预测并分配后续时间间隔所需的现场可编程门阵列数量， n_{t+1}。Spork 预测 n_{t+1} 而不是 n_t ，以便有时间启动所需的现场可编程门阵列。

算法 1 逐区间 FPGA 分配算法。

过程分配器
  n_{t-1}: 前一时间间隔所需的FPGA工作线程数
  n_{t+1}: 预测下一个时间间隔所需的 FPGA工作线程数量
  $\mathbb{H} \leftarrow$ 用于跟踪 FPGA工作线程数量分布的直方图哈希映射在一个区间内所需的量，取决于两个区间之前的数值
  $\mathbb{L} \leftarrow$ 现场可编程门阵列工作节点平均生命周期的哈希映射，条件是已分配的工作节点数量
  $\mathscr{F}_{t-1} \leftarrow$ 前一时间间隔内现场可编程门阵列上的请求服务时间总和
  $\mathscr{C}_{t-1} \leftarrow$前一时间间隔内各 CPU 上请求服务时间的总和
  $n_{t-1} \leftarrow$ 所需 FPGA 数量($\mathscr{F}_{t-1}$, $\mathscr{C}_{t-1}$)
  $\mathbb{H}[n_{t-3}].add(n_{t-1})$ ▷ 更新直方图
  $n_{t+1} \leftarrow$ 预测现场可编程门阵列($\mathbb{H}, \mathbb{L}, n_{t-1}$)
  分配现场可编程门阵列(n_{t+1})
  休眠(T_s)并重复 ▷ $T_s = A_f$
函数 所需 FPGA数量($\mathscr{F}$, $\mathscr{C}$)
  $\lambda \leftarrow \mathscr{F} + \mathscr{C}/S$ ▷ 请求大小的加权和
  $n \leftarrow \lfloor \frac{\lambda}{T_s} \rfloor$
  如果 λ 模 $T_s > T_b$ **那么** ▷ 来自公式 1
    $n \leftarrow n + 1$
  return n

Spork 通过估算在该时间间隔内，为了最优地满足请求总需求所需的现场可编程门阵列工作线程数量，来计算 n_{t-1}。为了估算请求总需求，Spork 将前一时间间隔内现场可编程门阵列和中央处理器工作节点上的请求大小相加，并按现场可编程门阵列加速比 S 加权。随后，它将总量除以时间间隔长度，以计算 n_{t-1}。Spork 根据盈亏平衡阈值 T_b 向上取整该估算值；该阈值是指在现场可编程门阵列上运行请求比使用中央处理器能效更高的服务时间界限。 T_b 定义如下：

$$T_b B_c = \frac{T_b}{S} B_f + \left(T_s - \frac{T_b}{S}\right) I_f, \tag{1}$$

其中， B_f、 B_c 和 I_f 分别是现场可编程门阵列和中央处理器在忙碌和空闲状态下的功率消耗。我们假设CPU 工作线程产生的空闲能量开销可以忽略，因为它们仅被短暂分配来服务请求突发。

Spork 随后使用 n_{t-1} 来预测下一时间间隔所需的现场可编程门阵列数量， n_{t+1}，具体通过调用 算法2。预测器使用 $\mathbb{H}$，一个历史工作节点数量的直方图哈希映射，以前两个时间间隔所需的工作节点数量为条件（因为现场可编程门阵列分配需要一个时间间隔）。映射条目按一个时间间隔内所需的现场可编程门阵列工作线程数量索引；每个条目都是一个直方图，用于记录两个时间间隔后所需工作节点数量的分布。直方图按每个时间间隔所需的现场可编程门阵列工作线程数量分箱，并跟踪迄今为止观测到这些数量的概率。每个时间间隔，分配器都会根据前一时间间隔所需的工作节点数量，更新（或初始化）$\mathbb{H}[n_{t-3}]$ 处的直方图， n_{t-1}。为计入启动开销，预测器还会使用 $\mathbb{L}$，一个以已分配工作节点数量为条件的现场可编程门阵列工作节点平均生命周期哈希映射。$\mathbb{L}$ 在工作节点取消分配时更新。

算法2 根据所需工作节点数量的历史分布，估计最具能效的现场可编程门阵列分配方案。

过程预测现场可编程门阵列($\mathbb{H}, \mathbb{L}, n_{t-1}$)
  n_{curr}：当前已分配的现场可编程门阵列数量
  n_{t+1}：下一个时间间隔的现场可编程门阵列工作节点预测数量
  如果 $n_{t-1} \notin \mathbb{H}$**则** ▷ 如果我们之前没有见过这个计数
    $n_{t+1} \leftarrow n_{t-1}$ ▷ 维持先前所需的工作节点数量
    return n_{t+1}
  $min_energy \leftarrow$UINT_MAX
  $hist \leftarrow \mathbb{H}[n_{t-1}]$ ▷ 使用直方图，位于 n_{t-1}
  对于所有 $\hat{n} \in hist.bins$，**执行** ▷ 工作节点数量的候选值
    $energy \leftarrow 0$
    $n_{new} \leftarrow 0$
    当 $\hat{n} > n_{curr} + n_{new}$ **时，执行** ▷ 计入摊销后的启动开销
      $avg_life \leftarrow \mathbb{L}[n_{curr} + n_{new}]$
      $avg_epochs \leftarrow \lceil avg_life/T_s \rceil$
      $energy \leftarrow energy + B_f A_f / avg_epochs$
      $n_{new} \leftarrow n_{new} + 1$
    对所有 $\{n, prob\} \in hist$**执行** ▷ 遍历分布
      如果 $\hat{n} > n$，**则** ▷ 过度分配
        $idle_energy \leftarrow (\hat{n} - n) I_f$
        $busy_energy \leftarrow n B_f$
      否则，如果 $\hat{n} < n$，**则** ▷ 分配不足
        $idle_energy \leftarrow 0$
        $busy_energy \leftarrow \hat{n} B_f + (n - \hat{n}) SB_c$
      $energy \leftarrow energy + prob \cdot (idle_energy + busy_energy)$
    如果 $energy < min_energy$ **则** ▷ 选择$\hat{n}$使其最小化 $energy$
      $min_energy \leftarrow energy$
      $n_{t+1} \leftarrow \hat{n}$
  返回 n_{t+1}

算法2 使用 $\mathbb{H}[n_{t-1}]$ 来预测 n_{t+1}，使该时间间隔内的预期能耗最小化。如果在 $\mathbb{H}[n_{t-1}]$ 处不存在直方图（即我们第一次看到该工作节点数量值），预测器会维持与前一时间间隔相同的分配。如果存在直方图，该算法会遍历 n_{t+1} 的候选值，这些值取自直方图分箱（或其间的值域）。它会找到 n_{t+1}，使预期能耗最小化；预期能耗定义为整个条件分布中忙碌和空闲能耗的加权和。如果候选值高于当前分配的现场可编程门阵列数量， n_{curr}，则给定 $\mathbb{L}$ 所对应的预期工作节点生命周期，算法还会计入一个时间间隔的摊销启动能耗。对于每个直方图条目（即预期工作节点数量及其出现概率），如果候选数量属于过度分配，则会因所使用的现场可编程门阵列而产生忙碌能耗开销以及额外现场可编程门阵列带来的空闲能量开销；如果分配不足，则只会产生所有现场可编程门阵列的忙时能量开销，以及派发路径上额外分配的中央处理器所产生的开销。该算法通过将每个预期工作节点数量对应的贡献乘以其发生概率，对分布中的预期能量开销求和。最后，选择使总体预期能耗最低的候选值。为减少开销，我们会缓存结果，并仅在直方图更新时延迟重新计算。

4.3 RequestDispatch 与快速分配

Alg. 3 Spork 按截止时间顺序处理请求，以满足延迟要求。它会估算请求大小并跟踪工作器完成时间，确保只有在请求能够按时完成时才将其分配给工作节点。如果没有任何可用工作器能够满足截止时间要求，调度器就会启动一个新的 CPU 工作线程，并将待处理请求 [18, 62, 75] 分配给它。

算法 3 优先效率的请求分派算法。

过程调度器
  $Q \leftarrow$ 请求队列，按截止时间排序
  适用于所有 $r \in Q$**的执行操作**
    $w \leftarrow$查找可用工作节点(r)
    如果 $w == null$ **那么**
      启动新的中央处理器 工作线程 c并将 r分配给它
    else
      将 r 分配给 w
  等待新请求到达，然后重复
函数FindAvailableWorker(r)
  $\beta_y \leftarrow$ 繁忙工作节点列表，按负载递减排序
  $\iota_y \leftarrow$ 空闲工作者列表，按空闲时长递增排序
  $\alpha_y \leftarrow$ 待分配工作节点列表，按排队负载递减排序
  对于所有 $y \in \{f, c\}$ **执行** ▷ 高效工作节点排序（现场可编程门阵列、中央处理器）
    对于所有 $w \in \{\beta_y, \iota_y, \alpha_y\}$ **执行**
      如果可满足截止期限(w， r)**那么**
        return w
  return $null$

请求调度也会影响混合平台的能耗，因为它决定了：（1）在现场可编程门阵列还是中央处理器上运行请求；（2）是否能快速回收空闲工作节点。我们的调度器将 AutoScale 基于索引的调度方案扩展到混合工作节点 [27]。具体而言，只要目标工作节点预计能够满足截止时间要求，就按工作节点效率顺序派发请求。在同类型工作节点组成的资源池中，它依次优先选择：（1）最繁忙的工作节点（即负载最高的工作节点）；（2）空闲时间最短的工作节点；（3）如果工作节点当前正在启动，则选择排队负载最高的工作节点。该策略通过将工作集中到预计最晚才会被释放的工作节点上，最大限度地提高效率，从而尽可能快速地回收其他工作节点。

4.4 权衡能量与成本

成本优化变体 Spork 在两个关键方面有所不同：(1) 它在 算法1 中使用基于成本的盈亏平衡阈值，该阈值为 $T_b = \frac{T_s C_f}{SC_c}$，以及 (2) 在考虑 算法2中的工作节点分配时，最小化总体成本，而不是能耗。

Spork 通过最小化预期能量和成本的可配置加权和，在能量与成本之间进行权衡。通常，过度分配现场可编程门阵列会大幅增加成本，但只会略微增加能量。

由于现场可编程门阵列价格昂贵但功率较低，因此使用量方面，现场可编程门阵列分配不足会大幅增加能耗，但只会略微增加成本，因为这部分缺口必须由效率低但价格便宜的中央处理器来填补。因此，运营商可以根据自身的目标进一步定制预测器。

4.5 局限性与讨论

Spork 假设有足够的工作节点可供扩容以应对到达负载；它不会针对稀缺资源进行优化。此外，算法 2 中能耗最优的 FPGA 数量计算算法 2 忽略了请求截止时间。考虑截止时间的 FPGA 分配可以提高 Spork 的能效（例如，如果截止时间足够宽裕，较少的 FPGA 就能处理相同的负载）；不过，对于我们的目标延迟敏感型应用，截止时间通常非常紧迫。我们将这些扩展留待未来工作。

我们对 Spork 调度器的描述也假设请求的执行时长已知。这是一种简化。有些领域（如机器学习推理）的请求大小非常稳定，可以预先进行分析 [30]。如果请求大小未知，我们可以采用先前研究中的在线请求大小估算器，因为已有研究表明，这些估算器对延迟敏感型应用是有效的 [14, 40, 45]。请注意，如果请求大小服从某种概率分布，Spork 只能满足概率性的服务等级目标（SLO），而不能保证硬性截止时间 [55]。

为简化说明，我们基于现场可编程门阵列与 CPU 之间现有的差异（例如启动时间存在巨大差距）对 Spork 进行了描述。然而，根据 GPU 和其他加速器的特性，也可以将其推广到任意工作节点。

5 评估

使用Spork，我们在大量工作负载和工作节点配置下比较混合平台与同构平台之间的权衡。我们展示了：

- Spork 在合理的应用与工作节点配置下运行生产环境云工作负载时，能量和成本最高分别降低 1.53× 和 2.6×，相较于纯 FPGA 平台最多减少 1.53× lower energyandcostsup to 2.6× less。
- Spork 变体可以有效地在能效与成本之间进行权衡；由于优先进行 FPGA分配，并提供最大化FPGA 利用率的请求分发策略，它们的表现优于现有的成本优化混合调度器。
- 混合平台能够以合理的成本，在广泛的配置范围内发挥 FP-GA的能效优势。对于请求较短且工作负载突发性中等至高的应用程序，它们尤其有效。

5.1 方法

我们使用 Cython 和 C++ 实现了一个离散事件模拟器，用于评估混合平台和同构平台。本节介绍实验的设置方式以及模拟器的配置方法。

工作节点。 中央处理器和现场可编程门阵列工作节点的参数设置基于 表 6，该调查见 § 2.2。我们假设工作节点在启动和关闭期间消耗忙碌功率。我们的默认设置反映了当前趋势：现场可编程门阵列工作节点启动较慢、能量消耗较少，而且单个实例的成本高于中央处理器。我们注意到，尽管这些参数看似合理，但对于任何特定应用程序，工作节点之间的成本和效率权衡都会有所不同。

	CPU 工作节点	现场可编程门阵列工作节点
启动延迟	5ms	*1秒，10秒，60秒，100秒*
降速延迟	5ms	100毫秒
相对加速比	1×	*1×*, 2×, 4×
忙碌功率	150W	*25W*，50W，*100W*
空闲功耗	*10W*，30W，*50W*	*10W*，20W，*30W*
按比例计算的成本	$0.668/小时	$0.982/小时

表6：工作节点配置。未使用斜体的值为默认值。

请求大小	Azure Functions	阿里巴巴 μ服务
短（10毫秒–100毫秒）	13	99
中等（100毫秒–1秒）	101	31
长（1秒–10秒）	241	N/A

表 7：根据我们的请求大小区间，从生产环境跟踪数据中评估的高需求应用数量。

工作负载。 我们评估 Spork 在生产环境跟踪数据和合成跟踪数据上。虽然没有公开数据集记录现场可编程门阵列的使用情况，我们重新利用了两个基于中央处理器的生产环境跟踪数据，如 表 7所示。Azure 跟踪数据由其公有云平台上的无服务器函数调用组成 [75]。阿里巴巴跟踪数据由生产微服务的远程过程调用组成 [51]。两个数据集都显示出极为倾斜的计算需求：不到 25% 的应用程序在任何时刻需要一个以上的工作节点，但它们占计算需求的 94% 以上。我们专注于这一子集，因为其他应用程序负载较低，最适合在 CPU 上运行。两个数据集都提供请求大小和每分钟请求到达率。我们使用请求到达率生成时变泊松到达间隔的两小时跟踪数据，并假设速率在每分钟内线性变化。我们评估适合以下请求大小区间的应用程序：短请求（10毫秒–100毫秒）、中等请求（100毫秒–1秒）和长请求（1秒–10秒）。

此外，我们通过三个步骤生成用于敏感性分析的合成跟踪数据。首先，从前述三个区间之一随机抽取请求大小（针对一个 CPU 工作线程）。接着，根据所选请求大小，基于自相似分布生成每分钟的请求到达率，使得平均需要 100 个 CPU 工作线程来处理负载。我们使用 b 模型 [87], 生成自相似速率跟踪数据，其中偏置参数表示突发程度，范围从 0.5（均匀负载）到 0.75（高度可变）。最后，我们使用这些基于速率的跟踪数据生成随时间变化的泊松到达间隔，与上文类似。除非另有说明，我们使用时长为两小时、请求大小较短且突发性为 0.6 的跟踪数据。

基线。 我们将 Spork 的能耗优化（SporkE）、成本优化（SporkC）和均衡（SporkB）变体与以下同构基线进行比较：(1) CPU 动态是一种基于无服务器框架和 AutoScale [27] 的仅 CPU 反应式调度器，它利用快速启动来应对突发负载（相当于仅使用 CPU 工作器的 Spork）。(2) FPGA 静态拥有完美的工作负载信息，并预先分配恰好足以应对峰值负载的现场可编程门阵列；它代表了静态配置的纯 FPGA 平台的最佳情况 [16, 65, 73]。(3) FPGA-dynamic 是一种纯 FPGA 反应式调度器，它在满足当前负载所需的最少现场可编程门阵列数量之上维持固定的额外容量 [4, 27, 72]。对于每条跟踪，FPGA-dynamic 分配满足要求的最小额外容量

基于连续时间间隔之间已知请求速率最大差值的整数倍确定请求截止时间。

我们还将 Spork 与 MArk 进行比较 [93], 。MArk 是一种最先进的、针对成本优化的 CPU-GPU 混合调度器。与 Spork 一样，MArk 将预测式与反应式工作节点管理相结合，并依赖请求大小估算来满足截止时间（否则，它会提供概率性 SLO [55]）。MArk 与 Spork 在以下几个关键方面有所不同。MArk：(1) 仅针对成本进行优化；(2) 使用一种速度较慢、数据密集型的 LSTM 模型进行预测 [31], ；(3) 预测多个时间间隔内细粒度的请求速率（这是一个颇具挑战性的预测问题），而非单个时间间隔内粗粒度的工作节点数量；(4) 使用轮询式请求派发，这种方式效率低下，并且在工作负载强度下降时更难回收工作节点。MArk 的性能在很大程度上取决于其预测器的准确性，而这并非其工作的目标 [93]。为给予其应有的考虑，并聚焦于两种调度方法之间的关键差异，我们评估了一个理想化版本的 MArk，`MArk-ideal`，它能够完美预测未来两个时间间隔内的工作负载。我们也同样评估了`SporkE-ideal`和SporkC-理想，两者都能完美预测下一个时间间隔内的工作负载，且不计入启动开销。

所有调度器都会在分配时长内让工作节点保持空闲，之后才将其关闭，以预期未来的请求 [27, 44, 75]。周期性调度器使用长度等于FPGA启动时间的调度区间。我们运行非理想的 Spork变体时，不会预热其预测器；在生产环境部署中，预测器可以随着时间推移学习工作负载分布，从而取得更好的结果。

指标。我们报告各调度器相对于理想化最优情况的纯 FPGA 平台的能量效率和成本，该平台采用默认参数，且仅产生计算成本（即空闲和启动开销均为零）。我们还确保请求满足截止时间，截止时间定义为 10× 请求大小。合成工作负载实验结果取十条随机生成轨迹的平均值，而生产环境工作负载仅运行一次。我们展示部分结果。

5.2 生产环境工作负载

表8 展示了各种调度器在默认配置下运行 Azure 和阿里巴巴跟踪数据时的总体能效和成本。我们发现，所有 `Spork` 变体在所有指标和跟踪数据上都优于同质平台。具体而言，对于具有短请求的 Azure 跟踪数据，`SporkE` 的成本与 CPU 动态大致相同，但能效高出 2×；此外，其能效最高可提高 53×，成本最高可降低 14×，低于FPGA 静态 和 `FPGA-dynamic`，从而实现两全其美。`SporkE`的效率和成本改进同样适用于阿里巴巴跟踪数据，但由于工作负载的突发性较低，其相对于现场可编程门阵列的优势有所减弱。具体而言，`SporkE` 比 CPU 动态 便宜 16×，且其效率提高16×，成本比 43× 低 FPGA 静态。对于中等请求大小，收益也较低。例如，在 Azure 跟踪数据上，`SporkE` 的能效提高15×，成本比 4× 低 `FPGA-dynamic`。

在混合调度器中，所有Spork变体都严格优于MArk-ideal，尽管它们开始时没有任何先前的工作负载信息。值得注意的是，尽管 MArk-ideal和SporkC 都经过成本优化，但后者提供的能量高出 1.1–2×

短请求。	Azure Functions（13 个应用）		阿里巴巴 μ服务（99 个应用）	
调度器	能量效率	相对 Cost	能量效率	相对的 Cost
CPU 动态	16.5%	1.35×	16.6%	1.34×
FPGA 静态	54.4%	3.08×	79.4%	1.64×
FPGA-dynamic	56.2%	2.87×	77.7%	1.68×
MArk-ideal	36.2%	1.41×	62.2%	1.14×
SporkC	73.5%	1.17×	84.8%	1.08×
SporkB	82.4%	1.23×	91.9%	1.12×
SporkE	86.2%	1.34×	92.8%	1.15×
SporkC-理想	76.6%	1.19×	90%	1.1×
SporkE-ideal	87.2%	1.31×	93.1%	1.14×

(a) 短请求大小。

中等请求	Azure Functions（101 个应用）		阿里巴巴 μ服务（31 个应用）	
调度器	能量效率	相对 Cost	能量效率	相对 Cost
CPU 动态	16.6%	1.33×	16.6%	1.34×
FPGA 静态	72.2%	1.96×	79.7%	1.63×
FPGA-dynamic	78.8%	1.63×	77.6%	1.67×
MArk-ideal	74.8%	1.11×	62.2%	1.16×
SporkC	81.2%	1.05×	79.5%	1.05×
SporkB	87.6%	1.09×	86.4%	1.07×
SporkE	91%	1.17×	92.1%	1.16×
SporkC-理想	92.4%	1.06×	88.5%	1.06×
SporkE-ideal	94.4%	1.09×	93%	1.13×

(b) 中等请求大小。

表 8：短请求大小和中等请求大小下生产环境跟踪数据的能效、成本和截止期限违约。所有应用程序的能耗和成本均汇总后报告，并以理想化的纯 FPGA 平台为参照。

追踪（请求大小）	轮询 [93]	索引打包 [27]	Spork
Azure（短）	60.6%	85.5%	86.2%
Azure（中等）	83.6%	81.9%	91%
Azure（长）	85.6%	84.2%	89.9%
阿里巴巴（短）	76.9%	88.8%	92.8%
阿里巴巴（中型）	85.7%	91.3%	92.1%

表9：在 SporkE 的工作节点分配逻辑下，不同派发策略对生产环境工作负载能效的影响。

效率，同时成本低 1.06–1.2×。SporkC-理想由于预测完美，进一步提升了SporkC的优势。我们发现，在所有轨迹上，Spork 变体都能适当地在能效与成本之间进行权衡。SporkE的能效最高可比SporkC 高1.17×，但成本最高也高出 1.14×。SporkB 在这两项指标上都达到了折中。

我们还注意到，Spork 的简单预测机制在实践中表现良好。无论从能效还是成本来看，SporkE 和 SporkC 始终与各自的理想版本十分接近。直观而言，当 FPGA 工作节点比中央处理器更节能（我们的默认配置）时，如果仅在中央处理器上执行 1% 的请求，与理想化的纯 FPGA 平台相比，能效就会降低 ~5%，其中不考虑启动和空闲开销。我们的结果表明，对于真实工作负载，Spork 会在现场可编程门阵列上处理大部分请求，并最大限度地提高其利用率。

5.3 理解 Spork 的功效

Spork 优于 MArk 的一个关键原因在于其请求分发策略，该策略专为混合平台设计。表9比较了 Spork 调度器与 MArk 轮询调度器的能效 [93] 以及 AutoScale 索引打包分发器的能效 [27]

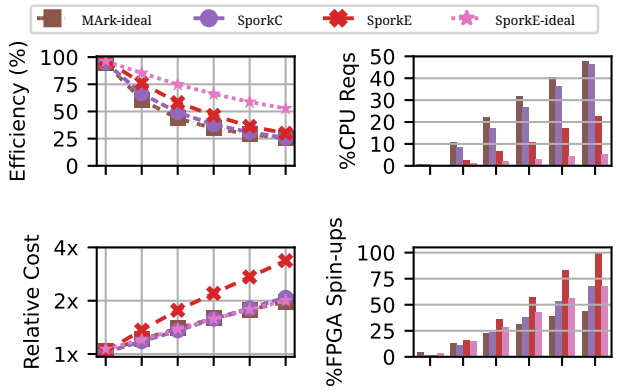


图 4：不同突发性条件下，Spork 与 MArk 的能效和成本权衡 [93]，以及 60 秒 现场可编程门阵列快速启动（左）。由 CPU 工作线程执行的请求百分比，以及按各调度器下的最大值归一化后的现场可编程门阵列快速启动总次数（右）。Spork-ideal 和MArk-ideal 具有完美的预测。

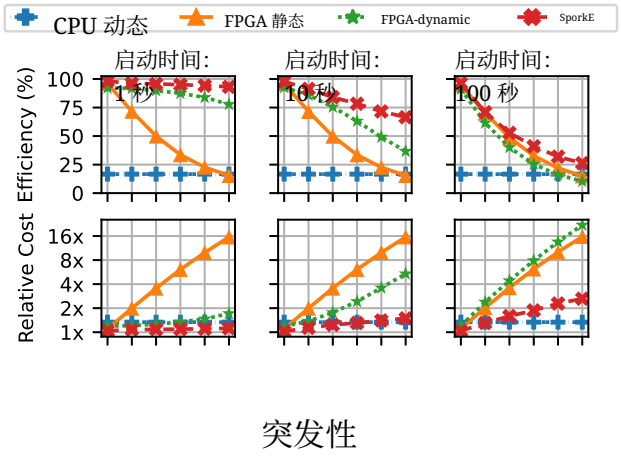


图5：对工作负载突发性和 FPGA 启动成本的敏感性。能效和成本均以采用 表6中默认参数的理想化仅FPGA 基线为参照进行报告。每个数据点均取 10 次轨迹运行的平均值。

在生产环境工作负载下。为保持一致，所有配置均采用 SporkE 的工作节点分配逻辑运行。我们发现，Spork 的调度器相比轮询能效最高可提升 42%，相比基于索引的派发能效最高可提升 11%。此前的两种策略都存在缺陷：轮询会将请求均匀分配给 CPU 和现场可编程门阵列，而且很少让工作节点空闲；索引打包则经常将请求派发给繁忙但效率低下的 CPU 工作线程，而现场可编程门阵列却处于空闲状态。相比之下，Spork 的调度器可最大限度提高现场可编程门阵列利用率，并帮助快速回收工作节点，从而实现高效率。

接下来，我们使用合成跟踪数据来了解预测误差的影响。图 4（左）评估了MArk-ideal 与 Spork 变体在突发性不断增加时的能效和成本权衡。突发性越高，Spork 的预测效果越差。更长的调度区间也会增加 Spork 准确预测的难度，因此我们将调度间隔长度（以及现场可编程门阵列快速启动时间）设为 60 秒。我们发现：(1) SporkC 和SporkE-ideal 的能效最高可比13× 和1× MArk-ideal 高，同时成本大致相同；(2) SporkE 的表现优于 MArk-ideal，能效最高可提升34×，但在中等突发性下成本会增加38×；(3) 在高突发性下，SporkE 与SporkE-ideal 在能效和成本方面最多相差75×。

为探究其中原因，我们针对每个调度器绘制了在整个工作负载中于 CPU 上执行的请求总数百分比，以及所分配现场可编程门阵列的数量（以最大值归一化），如 图 4（右）所示。我们发现，MArk-ideal 和 SporkC 的成本较低，这是因为它们分配的现场可编程门阵列更少，并且会在 CPU 上执行多达一半的请求；然而，大量使用 CPU 会降低它们的能效。SporkE 仅在 CPU 上执行最多 22% 的请求，从而提供更高的能效；然而，误预测导致其分配的现场可编程门阵列几乎比 3× MArk-ideal 多 2 倍，从而带来更高的成本。SporkE-ideal的完美预测进一步提升了其效率优势；它执行的请求少于

将 CPU 上的请求的 ~5% 用于处理瞬时突发流量。尽管 SporkE-ideal 分配的现场可编程门阵列最多比 MArk-ideal 多 1.5×，但其总体成本相近

因为这些分配规模恰当，并配有能够最大限度提高其利用率的调度器。

5.4 敏感性分析

我们通过在多种配置下将 SporkE 与同构平台进行比较，量化了对我们的假设的敏感性。

工作负载突发性和 FPGA 启动成本。图 5 展示了工作负载突发性和 FPGA 启动时间变化对能效和成本的影响。我们观察到：(1) CPU-reactive在突发性较高时具有成本效益，而纯 FPGA 平台尽管存在过度配置，通常仍能效较高；(2) FPGA-dynamic相较于 FPGA 静态的优势会随着 FPGA 启动时间的增加而减弱；(3)SporkE 在能效方面优于或不逊于其他方案，同时比纯 FPGA 平台便宜得多；(4) 更高的突发性会增强 SporkE相较于纯 FPGA 平台的效率和成本优势；(5) 更长的 FPGA 启动时间会削弱 SporkE的能效优势，但会扩大其相较于 FPGA-dynamic的成本优势。以下将对此进行详细说明。

在同构平台中，CPU 动态 在中到高突发性下比纯 FPGA 平台更具成本效益——最多便宜 11。8× 比 FPGA 静态便宜。在相反的情况下，FPGA 静态和 FPGA-dynamic尽管存在过度配置，能效仍更高。随着突发性和快速启动成本上升，它们的优势便会消失，此时仅使用 CPU 更可取。这些结果与我们在 § 3中观察到的趋势一致。在纯 FPGA 平台中，FPGA-dynamic 在快速启动时间较短时显著优于 FPGA 静态 。快速启动时间较长时，FPGA-dynamic 的表现较差，因为它需要更大的余量来应对突发负载。与传统自动扩缩系统一样，[4, 56, 72], FPGA-dynamic 会在负载波动时维持这一余量，因此有时会分配比 FPGA 静态更多的工作节点。相比之下，FPGA 静态 只产生少量的一次性快速启动成本；由于掌握了精准的工作负载信息，它还受益于精确的资源配置。

即使没有事先掌握工作负载信息，SporkE 相较于同构平台也能显著提升能效并带来成本效益

（请求大小除外）。即使突发性很高，SporkE 也比 CPU-reactive 至多便宜 1.2×，能效高 5.6×，但快速启动时间较长时，这一成本优势便不复存在。SporkE 的效率也比 FPGA-dynamic 高至多 2.5×（便宜 8.4×），比 FPGA-static 高至多 6×（便宜 13.9×）。

SporkE 相较于纯 FPGA 平台的优势会随着突发性增强而扩大。例如，在突发性中等且快速启动时间为 10 秒时，SporkE 的能效比 FPGA-dynamic 高 1.1×，成本低 1.4×；在突发性较高时，这一差距分别扩大至能效高 1.8×、成本低 3.6×。纯 FPGA 平台需要分配昂贵的现场可编程门阵列，但这些 FPGA 仅被短暂使用，因此表现不佳。

与 FPGA-dynamic 类似，随着现场可编程门阵列启动时间增加，SporkE 的能效优势会减弱，原因有三：（1）启动成本不断累积；（2）在较长时间间隔内发生请求突发的概率更高；（3）预测变得更加困难，尤其是预测器可用于学习的样本更少。不过，随着快速启动时间延长，SporkE 的成本会比 FPGA-dynamic 低得多（最高可达 8.4×），因为后者需要预留大量 FPGA 容量，因而处境更差。与突发性中等、启动时间为 100 秒的 FPGA-static 相比，SporkE 的能效仅高 1.08×，但成本仍低 2.3×。

工作节点效率。 图6展示了随着现场可编程门阵列在服务请求方面变得比 CPU 更快且功率效率更高，其能效和成本影响。[2] 在所有情况下， SporkE 都保持了相对于两种仅FPGA替代方案的成本优势，同时提供了相似或更高的能效。我们还做出以下补充观察。

首先，现场可编程门阵列处理速度的加速比相比混合平台，能为纯FPGA 平台带来更大的优势。 4× 现场可编程门阵列加速比可使纯 FPGA 平台的能效和成本获得近乎线性的改善。由于SporkE将部分工作负载运行在CPU工作器上，其改善幅度较小（3.4– 3.7×）。更高的加速比会使 SporkE 的误预测相对更加昂贵。由于成本大幅降低，它们甚至可以使纯 FPGA平台相较于仅 CPU 平台更具绝对优势——例如，在 4× 加速比下，FPGA-dynamic 的成本最多可比 CPU 动态低 1.47×。

其次，对于纯FPGA 平台而言，优化现场可编程门阵列的繁忙功耗效率会带来递减的收益，因为空闲功耗开始占主导。例如，现场可编程门阵列忙碌功率降低 4× 会带来
仅使 FPGA 静态的能耗降低 ~2×，而 SporkE 的能耗则降低了超过 3×，因为它能更好地利用现场可编程门阵列。当现场可编程门阵列的忙碌功率为 100W 时，空闲状态占 FPGA 静态总能耗的 33%；当功率降至 25W 时，这一比例上升到 66% 以上。对于 SporkE，空闲能耗占比仅从 4% 上升到 13%。随着忙碌功率的降低，SporkE 与 FPGA-dynamic 之间的效率差距也会扩大（从 1.04× 增至 1.12×），因为后者承担的空闲开销越来越大。总的来说，提升现场可编程门阵列的速度比优化功耗更有帮助，因为使用更少的工作节点就能处理相同的工作负载。

请求大小。 图 7 考察了应用程序请求大小（以及相应的截止时间）变化所带来的影响。较长的请求和截止时间能显著提升仅 FPGA 平台的表现，因为它们可以提高工作节点利用率，并减轻较慢的启动时间带来的影响。 FPGA 静态的能效最高提升了 16%

[2]在我们的模型中，功率效率的提升不会影响成本，因为我们采用固定的云服务价格。根据总拥有成本（TCO）分析，即使将运营成本纳入考量，趋势仍然相似 [7]。

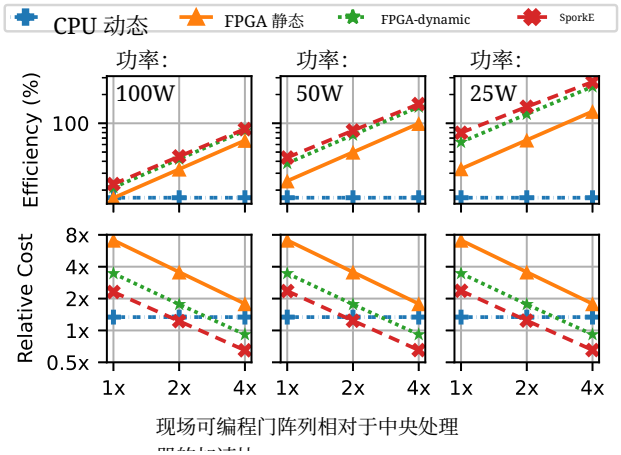


图6：对现场可编程门阵列性能和忙碌功率消耗的敏感性。请注意，两个坐标轴均采用对数刻度。

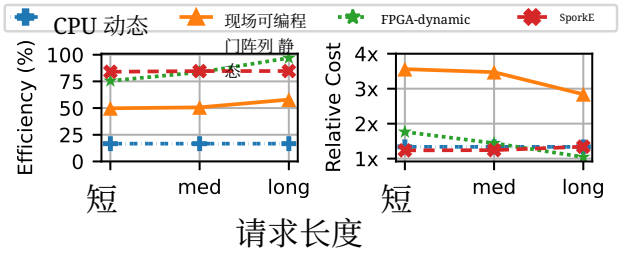


图7：对请求大小的敏感性。我们考察短请求（10毫秒– 100毫秒）、中等请求（100毫秒–1秒）和长请求（1秒–10秒）。截止时间为 10× 请求大小。

在请求较长时，由于服务相同负载所需的工作节点更少，因此其表现更好。 FPGA-dynamic的能效提高了 1。3×，成本降低了 1。7×，因为它需要维持的备用容量较小。事实上，对于超过 1秒 的请求，由于截止时间长于 FPGA 启动时间，因此无需预留备用容量。在这种情况下，纯反应式调度器表现良好，因为它可以自动适应负载，同时留有充足余地来满足截止时间。Spork的优势会随着请求变长而减弱，因为它在分配 FPGA 时不会考虑截止时间（§ 4.2）。

6 相关工作

据我们所知，我们首次展示了混合FPGA-CPU平台能够以较低成本，为延迟敏感型和突发型应用程序充分发挥现场可编程门阵列的能效优势，并构建了一个能够权衡这两项指标的混合调度器。

数据中心FPGA。 现场可编程门阵列通常被捆绑在虚拟机抽象之下，并作为外围设备进行管理 [16, 46, 52, 73, 92]。有些工作将其提升为一等计算资源，并通过 Open-Stack 等数据中心编排器直接管理 [1, 9, 22, 69, 81, 82, 88, 89]，；另一些工作则针对推理服务 [5, 15, 32]、键值存储 [39, 48]，或大数据处理 [34] 等特定应用程序定制现场可编程门阵列的使用方式。尽管这些工作支持数据中心中的多种现场可编程门阵列使用场景，但它们无法在延迟敏感且突发性强的应用程序之间高效共享现场可编程门阵列，因此需要过度配置。最近，现场可编程门阵列已……

已被纳入无服务器框架，从而能够在共享工作池上实现动态管理 [6, 20, 32, 35, 61]。与我们的设计最接近的可能是 Molecule [20], , 它在中央处理器和现场可编程门阵列上提供了无服务器抽象。Molecule 将小型应用程序的现场可编程门阵列启动时间缩短至几秒，并帮助在多个实例之间共享现场可编程门阵列，以提高效率。然而，它没有提供调度器来在不同后端上交替运行请求，从而优化整体效率和成本。

混合计算。以往研究主要关注 GPU-CPU 混合平台，其权衡取舍与 FPGA-CPU 平台不同。大多数工作面向机器学习推理服务，试图在满足延迟期限的同时优化成本和吞吐量 [60, 70, 90, 93]。它们通常采用请求批处理和模型变体等领域专用技术来提升性能。AlloX 面向深度学习训练等批处理作业，并提供一种调度算法，在优化性能的同时保障公平性 [49]。然而，这些工作均未考虑能效，也没有探讨在多种工作负载配置下，将混合平台用于延迟敏感型和突发型应用程序所带来的权衡取舍。

交互式调度。 调度器大致可分为反应式和预测式两类。反应式调度器会动态响应需求；当请求截止时间长于工作器启动时间时，其表现良好 [27, 62, 79, 83]；大多数无服务器调度器都属于这一类。预测式调度器试图主动调整工作节点分配，以满足应用服务级别目标 [29, 43, 77, 93] 。

Spork 兼顾这两方面，并提出一种新的预测机制来提升效率 [26, 93]。根据优化指标的不同，现有调度器中的请求分发机制要么均匀分配负载，以提高利用率并降低尾延迟 [63, 93] ，要么集中负载以提高效率 [27, 79]。我们将后一种方法扩展到混合平台。一些无服务器调度器试图通过预测 [71, 75, 90] 和缓存 [25] 来尽量降低冷启动（即快速启动）的影响。然而，减少冷启动的工作主要针对不常用的应用程序，而在现场可编程门阵列上运行这些应用程序并不高效。

7 结论

我们提出了一种 FPGA-CPU 混合计算框架，既能发挥 FPGA 的能效优势，又能降低延迟敏感型和突发型应用的成本。为实现这些优势，我们设计了 Spork——一种高效的混合调度器，可优化并权衡能效与成本。Spork 包含：（1）一个定期分配现场可编程门阵列的轻量级预测器；（2）一个能利用 CPU 快速启动的异构性感知请求分发器。我们的调度器充分考虑了现场可编程门阵列与 CPU 工作节点在效率、成本和启动时间方面的关键差异。我们使用生产环境和合成跟踪数据，在大量工作负载和工作节点配置下对 Spork 进行了评估。针对截止时间严格的云工作负载，结果表明，能效优化型 Spork 的能耗最高可降低 1.53×，成本可减少 2.14×，相较于纯 FPGA 平台优势显著。与理想化的 MArk（成本优化型混合调度器）相比，能效优化型 Spork 在成本相近的情况下，能效最高可提升 1.2–2.4×。

致谢

本研究由美国国家科学基金会 CNS-2104548 项目以及 VMware 和思科提供的研究资助支持。

参考文献

[1] Francois Abel、Jagath Weerasinghe、Christoph Hagleitner、Beat Weiss 和 Stephan Paredes。2017年。面向超大规模云服务商的 FPGA 平台。年度高性能互连研讨会（HOTI）。[2] Alexandru Agache、Marc Brooker、Alexandra Iordache、Anthony Liguori、Rolf Neugebauer、Phil Piwonka 和 Diana-Maria Popa。2020年。Firecracker：用于无服务器应用的轻量级虚拟化。USENIX 操作系统设计与实现研讨会（NSDI）。[3] Apache。OpenWhisk。检索于2021年5月14日，网址：https://openwhisk.apache.org/。[4] 微软 Azure。Azure Functions 高级计划：预热实例。检索于2021年5月16日，网址：https://docs.microsoft.com/en-us/azure/azure-functions/functions-premium-plan。[5] 微软 Azure。将机器学习模型部署到现场可编程门阵列。检索于2021年5月13日，网址：https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service。[6] Marco Bacis、Rolando Brondolin 和 Marco Santambrogio。2020年。BlastFunction：用于加速无服务器计算的现场可编程门阵列即服务系统。欧洲设计、自动化与测试会议（DATE）。[7] Luiz André Barroso、Urs Hölzle 和 Parthasarathy Ranganathan。2018年。数据中心即计算机：仓库规模计算机的设计。计算机体系结构综合讲座。[8] Andrew Boutros、Eriko Nurvitadhi、Rui Ma、Sergey Gribok、Zhipeng Zhao、James C. Hoe、Vaughn Betz 和 Martin Langhammer。2020年。超越峰值性能：比较人工智能优化型现场可编程门阵列与 GPU 的实际性能。国际现场可编程技术会议（FPT）。[9] Stuart Byma、J. Gregory Steffan、Hadi Bannazadeh、Alberto Leon Garcia 和 Paul Chow。2014年。云中的现场可编程门阵列：使用 OpenStack 引导虚拟化硬件加速器。国际现场可编程定制计算机研讨会（FCCM）。[10] Adrian Caulfield、Eric Chung、Andrew Putnam、Hari Angepat、Jeremy Fowers、Michael Haselman、Stephen Heil、Matt Humphrey、Puneet Kaur、Joo-Young Kim、Daniel Lo、Todd Massengill、Kalin Ovtcharov、Michael Papamichael、Lisa Woods、Sitaram Lanka、Derek Chiou 和 Doug Burger。2016年。云规模加速架构。国际微体系结构研讨会（MICRO）。[11] Doris Chen 和 Deshanand Singh。2013年。使用 OpenCL 进行分形视频压缩：对 CPU、GPU 和现场可编程门阵列作为加速平台的评估。亚洲及南太平洋设计自动化会议（ASP-DAC）。[12] Doris Chen 和 Deshanand Singh。2012年。使用 OpenCL 评估 CPU、GPU 和现场可编程门阵列在信息过滤方面的效率。国际现场可编程逻辑与应用会议（FPL）。[13] Tianqi Chen、Thierry Moreau、Ziheng Jiang、Lianmin Zheng、Eddie Yan、Haichen Shen、Meghan Cowan、Leyuan Wang、Yuwei Hu、Luis Ceze、Carlos Guestrin 和 Arvind Krishnamurthy。2018年。TVM：用于深度学习的自动化端到端优化编译器。USENIX 操作系统设计与实现研讨会（OSDI）。[14] Chih-Hsun Chou、Laxmi N Bhuyan 和 Daniel Wong。2019年。μDPM：微秒时代的动态功耗管理。国际高性能计算机体系结构研讨会（HPCA）。[15] Eric Chung、Jeremy Fowers、Kalin Ovtcharov、Michael Papamichael、Adrian Caulfield、Todd Massengill、Ming Liu、Daniel Lo、Shlomi Alkalay、Michael Haselman、Maleen Abeydeera、Logan Adams、Hari Angepat、Christian Boehn、Derek Chiou、Oren Firestein、Alessandro Forin、Kang Su Gatlin、Mahdi Ghandi、Stephen Heil、Kyle Holohan、Ahmad El Husseini、Tamas Juhasz、Kara Kagi、Ratna K. Kovvuri、Sitaram Lanka、Friedel van Megen、Dima Mukhortov、Prerak Patel、Brandon Perez、Amanda Rapsang、Steven Reinhardt、Bita Rouhani、Adam Sapek、Raja Seera、Sangeetha S hekar、Balaji Sridharan、Gabriel Weisz、Lisa Woods、Phillip Yi Xiao、Dan Zhang、Ritchie Zhao 和 Doug Burger。2018年。借助 Project Brainwave 在数据中心规模上实时提供深度神经网络服务。IEEE Micro。[16] 阿里云。深入了解阿里云 F3 现场可编程门阵列即服务实例。检索于2021年5月13日，网址：https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057。[17] 阿里云。价格计算器。检索于2023年1月13日，网址：https://www.alibabacloud.com/pricing-calculator。[18] Cloudflare。借助 Cloudflare Workers 消除冷启动。检索于2021年5月16日，网址：https://blog.cloudflare.com/eliminating-cold-starts-with-cloudflare-workers/。[19] Cloudflare。Workers。检索于2021年5月14日，网址：https://workers.cloudflare.com/。

[20] Dong Du、Qingyuan Liu、Xueqiang Jiang、Yubin Xia、Binyu Zang 和 Haibo Chen。2022年。异构计算机上的无服务器计算。国际编程语言与操作系统体系结构支持会议（ASPLOS）。[21] Dong Du、Tianyi Yu、Yubin Xia、Binyu Zang、Guanglu Yan、Chenggang Qin、Qixuan Wu 和 Haibo Chen。2020年。Catalyzer：通过无初始化引导实现无服务器计算的亚毫秒级启动。国际编程语言与操作系统体系结构支持会议（ASPLOS）。[22] Nariman Eskandari、Naif Tarafdar、Daniel Ly-Ma 和 Paul Chow。2019年。用于在数据中心部署现场可编程门阵列和 CPU 的模块化异构堆栈。国际现场可编程门阵列研讨会（FPGA）。[23] Jeremy Fowers、Kalin Ovtcharov、Michael Papamichael、Todd Massengill、Ming Liu、Daniel Lo、Shlomi Alkalay、Michael Haselman、Logan Adams、Mahdi Ghandi、Stephen Heil、Prerak Patel、Adam Sapek、Gabriel Weisz、Lisa Woods、Sitaram Lanka、Steven Reinhardt、Adrian Caulfield、Eric Chung 和 Doug Burger。2018年。用于实时人工智能的可配置云规模深度神经网络处理器。国际计算机体系结构研讨会（ISCA）。[24] Martin Fowler。微服务。检索于2021年5月16日，网址：https://martinfowler.com/articles/microservices.html。[25] Alexander Fuerst 和 Prateek Sharma。2021年。FaasCache：通过贪婪双重缓存保持无服务器计算持续运行。国际编程语言与操作系统体系结构支持会议（ASPLOS）。[26] Anshul Gandhi、Yuan Chen、Daniel Gmach、Martin Arlitt 和 Manish Marwah。2011年。通过混合资源供给最大限度地减少数据中心服务级别协议违约和功耗。国际绿色计算会议及研讨会。[27] Anshul Gandhi、Mor Harchol-Balter、Ram Raghunathan 和 Michael Kozuch。2012年。AutoScale：多层数据中心的动态稳健容量管理。ACM计算机系统汇刊（TOCS）。[28] CNCF无服务器工作组。CNCF无服务器白皮书 v1.0。检索于2020年12月23日，网址：https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/README.md。[29] Arpan Gujarati、Sameh Elnikety、Yuxiong He、Kathryn McKinley 和 Björn Brandenburg。2017年。Swayam：通过分布式自动扩缩容，以高资源效率满足机器学习推理服务的服务级别协议。国际中间件会议（Middleware）。[30] Arpan Gujarati、Reza Karimi、Safya Alzayat、Wei Hao、Antoine Kaufmann、Ymir Vigfusson 和 Jonathan Mace。2020年。像时钟一样服务深度神经网络：自底向上的性能可预测性。USENIX 操作系统设计与实现研讨会（OSDI）。[31] Sepp Hochreiter 和 Jürgen Schmidhuber。1997年。长短期记忆。神经计算。[32] Steve Hoover。1st-CLaaS。检索于2021年5月14日，网址：https://github.com/stevehoover/1st-CLaaS。[33] Mohammad Hosseinabady 和 Jose Luis Nunez-Yanez。2014年。混合 ARM-FPGA 设备中的运行时电源门控。国际现场可编程逻辑与应用会议（FPL）。[34] Muhuan Huang、Di Wu、Cody Hao Yu、Zhenman Fang、Matteo Interlandi、Tyson Condie 和 Jason Cong。2016年。支持在数据中心规模部署现场可编程门阵列加速器的编程与运行时系统。ACM云计算研讨会（SoCC）。[35] InAccel。FPGA管理器。检索于2021年5月13日，网址：https://inaccel.com/fpga-manager/。[36] 英特尔。oneAPI：异构计算的新时代。检索于2022年3月6日，网址：https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html。[37] 英特尔。Stratix® 10配置用户指南。检索于2021年5月15日，网址：https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-config.pdf。[38] 英特尔。Stratix® V器件数据手册。检索于2021年5月15日，网址：https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-v/stx5_53001.pdf。[39] Zsolt István、David Sidler 和 Gustavo Alonso。2017年。Caribou：智能分布式存储。VLDB基金会会刊。[40] Michael A. Iverson、Fusun Ozguner 和 Lee C. Potter。1999年。通过分析基准测试预测任务执行时间，以支持异构环境中的调度。异构计算研讨会（HCW）。[41] Zhipeng Jia 和 EmmettWitchel。2021年。Nightcore：面向延迟敏感交互式微服务的高效可扩展无服务器计算。国际编程语言与操作系统体系结构支持会议（ASPLOS）。[42] Eric Jonas、Johann Schleier-Smith、Vikram Sreekanti、Chia-Che Tsai、Anurag Khandelwal、Qifan Pu、Vaishaal Shankar、Joao Menezes Carreira、Karl Krauth、Neeraja Yadwadkar、Joseph Gonzalez、Raluca Ada Popa、Ion Stoica 和 David Patterson。2019年。简化云编程：伯克利对无服务器计算的观点。UCB/EECS-2019-3。加利福尼亚大学伯克利分校电气工程与计算机科学系。

[43] Ram Srivatsa Kannan、Lavanya Subramanian、Ashwin Raju、Jeongseob Ahn、Jason Mars 和 Lingjia Tang。2019。GrandSLAm：为微服务执行框架中的作业保障服务级别协议。欧洲计算机系统会议（EuroSys）。[44] Anna R. Karlin、Mark S. Manasse、Lyle A. McGeoch 和 Susan Owicki。1994。针对非均匀问题的竞争性随机算法。Algorithmica。[45] Harshad Kasture、Davide B Bartolini、Nathan Beckmann 和 Daniel Sanchez。2015。Rubik：面向延迟关键型系统的快速分析式功率管理。国际微体系结构研讨会（MICRO）。[46] Ahmed Khawaja、Joshua Landgraf、Rohith Prakash、Michael Wei、Eric Schkufza 和 Christopher Rossbach。2018。借助 AmorphOS 实现可重新配置架构的共享、保护与兼容性。USENIX 操作系统设计与实现研讨会（OSDI）。[47] Dario Korolija、Timothy Roscoe 和 Gustavo Alonso。2020。操作系统抽象在现场可编程门阵列上是否合理？USENIX 操作系统设计与实现研讨会（OSDI）。[48] Maysam Lavasani、Hari Angepat 和 Derek Chiou。2014。基于现场可编程门阵列的 Memcached 内联加速器。IEEE Computer Architecture Letters（CAL）。[49] Tan N. Le、Xiao Sun、Mosharaf Chowdhury 和 Zhenhua Liu。2020。AlloX：混合集群中的计算资源分配。欧洲计算机系统会议（EuroSys）。[50] Will E. Leland、Murad S. Taqqu、Walter Willinger 和 Daniel V. Wilson。1994。以太网流量的自相似特性（扩展版）。IEEE/ACM 网络学报（TON）。[51] Shutian Luo、Huanle Xu、Chengzhi Lu、Kejiang Ye、Guoyao Xu、Liping Zhang、Yu Ding、Jian He 和 Chengzhong Xu。2021。微服务依赖关系与性能分析：阿里巴巴跟踪数据分析。ACM云计算研讨会（SoCC）。[52] Jiacheng Ma、Gefei Zuo、Kevin Loughlin、Xiaohe Cheng、Yanqiang Liu、Abel Mulugeta Eneyew、Zhengwei Qi 和 Baris Kasikci。2020。面向共享内存现场可编程门阵列平台的虚拟机监控程序。国际编程语言与操作系统体系结构支持会议（ASPLOS）。[53] Filipe Manco、Costin Lupu、Florian Schmidt、Jose Mendes、Simon Kuenzer、Sumit Sati、Kenichi Yasukata、Costin Raiciu 和 Felipe Huici。2017。我的虚拟机比你的容器更轻量（也更安全）。操作系统原理研讨会（SOSP）。[54] Vaibhawa Mishra、Qianqiao Chen 和 Georgios Zervas。2016。REoN：一种通过网络实现可靠软件定义现场可编程门阵列部分重配置的协议。可重配置计算与现场可编程门阵列国际会议（ReConFig）。[55] Jeffrey Mogul 和 John Wilkes。2019。九个九还不够：云的有意义指标。操作系统热点研讨会（HotOS）。[56] Hiep Nguyen、Zhiming Shen、Xiaohui Gu、Sethuraman Subbiah 和 John Wilkes。2013。Agile：面向基础设施即服务的弹性分布式资源扩展。自主计算国际会议（ICAC）。[57] Tan Nguyen、Colin MacLean、Marco Siracusa、Douglas Doerfler、Nicholas J Wright 和 Samuel Williams。2022。基于现场可编程门阵列的高性能计算加速器：性能与能效评估。Concurrency and Computation: Practice and Experience。[58] Eriko Nurvitadhi、Jaewoong Sim、David Sheffield、Asit Mishra、Srivatsan Krishnan 和 Debbie Marr。2016。在分析服务器中加速循环神经网络：现场可编程门阵列、中央处理器、GPU 和 ASIC 的比较。国际现场可编程逻辑与应用会议（FPL）。[59] Eriko Nurvitadhi、Ganesh Venkatesh、Jaewoong Sim、Debbie Marr、Randy Huang、Jason Ong Gee Hock、Yeong Tat Liew、Krishnan Srivatsan、Duncan Moss、Suchit Subhaschandra 和 Guy Boudoukh。2017。现场可编程门阵列能否在加速新一代深度神经网络方面胜过 GPU？国际现场可编程门阵列研讨会（FPGA）。[60] NVIDIA。Triton 推理服务器。检索于2021年5月18日，网址：https://developer.nvidia.com/nvidia-triton-inference-server。[61] David Ojika、Ann Gordon-Ross、Herman Lam 和 Bhavesh Patel。2019。FaaM：微服务即现场可编程门阵列——数据压缩案例研究。EPJ Web of Conferences。[62] OpenFaaS。让无服务器函数化繁为简。检索于2021年5月14日，网址：https://www.openfaas.com/。[63] Amy Ousterhout、Joshua Fried、Jonathan Behrens、Adam Belay 和 Hari Balakrishnan。2019。Shenango：实现延迟敏感型数据中心工作负载的高中央处理器效率。USENIX 网络系统设计与实现研讨会（NSDI）。[64] Rekha Pitchumani、Shayna Frank 和 Ethan L. Miller。2015。在存储基准测试中生成真实的请求到达序列。大容量存储系统与技术研讨会（MSST）。[65] Andrew Putnam、Adrian Caulfield、Eric Chung、Derek Chiou、Kypros Con- stantinides、John Demme、Hadi Esmaeilzadeh、Jeremy Fowers、Gopi Prashanth Gopal、Jan Gray、Michael Haselman、Scott Hauck、Stephen Heil、Amir Hor- mati、Joo-Young Kim、Sitaram Lanka、James Larus、Eric Peterson、Simon Pope、Aaron Smith、Jason Thong、Phillip Yi Xiao 和 Doug Burger。2014。一种用于加速大规模数据中心服务的可重新配置架构。计算机体系结构国际研讨会（ISCA）。

[66] Murad Qasaimeh、Kristof Denolf、Jack Lo、Kees Vissers、Joseph Zambreno 和 Phillip Jones。2019。比较视觉内核的中央处理器、GPU 和现场可编程门阵列实现的能效。嵌入式软件与系统国际会议（ICESS）。[67] Weikang Qiao、Zhenman Fang、Mau-Chung Frank Chang 和 Jason Cong。2019。一种用于 Bzip2 数据压缩的基于现场可编程门阵列的 BWT 加速器。国际现场可编程定制计算机研讨会（FCCM）。[68] Burkhard Ringlein、Francois Abel、Alexander Ditter、Beat Weiss、Christoph Ha- gleitner 和 Dietmar Fey。2020。使用 MPI 与转译技术对可重新配置异构计算集群进行编程。异构高性能可重新配置计算国际研讨会（H2RC）。[69] Burkhard Ringlein、Francois Abel、Alexander Ditter、Beat Weiss、Christoph Hagleitner 和 Dietmar Fey。2019。使用部分重配置的云端网络连接型现场可编程门阵列系统架构。国际现场可编程逻辑与应用会议（FPL）。[70] Francisco Romero、Qian Li、Neeraja J Yadwadkar 和 Christos Kozyrakis。2021。INFaaS：自动化的无模型推理服务。USENIX 年度技术会议（ATC）。[71] Rohan Basu Roy、Tirthak Patel 和 Devesh Tiwari。2022。IceBreaker：借助异构性更好地预热无服务器函数。国际编程语言与操作系统体系结构支持会议（ASPLOS）。[72] Amazon Web Services。Amazon ECS 容量与扩展最佳实践指南。检索于2021年5月16日，网址：https://docs.aws.amazon.com/Amazon ECS/latest/bestpracticesguide/capacity-availability.html。[73] Amazon Web Services。EC2 F1 实例。检索于2021年5月13日，网址：https: // aws.amazon.com/ec2/instance-types/f1/。[74] Amazon Web Services。EC2 30, 2022 2按需实例定价。检索于 Apr.，网址：https://aws.amazon.com/ec / pricing/on-demand/。[75] Mohammad Shahrad、Rodrigo Fonseca、Íñigo Goiri、Gohar Chaudhry、Paul Batum、Jason Cooke、Eduardo Laureano、Colby Tresness、Mark Russinovich 和 Ricardo Bianchini。2020。真实环境中的无服务器：大型云服务提供商的无服务器工作负载分析与优化。USENIX 年度技术会议（ATC）。[76] Hardik Sharma、Jongse Park、Divya Mahajan、Emmanuel Amaro、Joon Kyung Kim、Chenkai Shao、Asit Mishra 和 Hadi Esmaeilzadeh。2016。从高层深度神经网络模型到现场可编程门阵列。国际微体系结构研讨会（MICRO）。[77] Arjun Singhvi、Kevin Houck、Arjun Balasubramanian、Mohammed Danish Shaikh、Shivaram Venkataraman 和 Aditya Akella。Archipelago：可扩展的低延迟无服务器平台。（2019）。[78] Rym Skhiri、Virginie Fresse、Jean Paul Jamont、Benoit Suffran 和 Jihene Malek。2019。从现场可编程门阵列到云支持，再到现场可编程门阵列云：现状。International Journal of Reconfigurable Computing。[79] Amoghvarsha Suresh 和 Anshul Gandhi。2019。FnSched：一种高效的无服务器函数调度器。无服务器计算国际研讨会。[80] Naif Tarafdar 和 Paul Chow。2019。libGalapagos：用于现场可编程门阵列与中央处理器异构应用程序原型设计和开发的软件环境。面向软件程序员的现场可编程门阵列国际研讨会（FSP Workshop）。[81] Naif Tarafdar、Thomas Lin、Eric Fukuda、Hadi Bannazadeh、Alberto Leon-Garcia 和 Paul Chow。2017。实现异构云数据中心中的灵活网络现场可编程门阵列集群。国际现场可编程门阵列研讨会（FPGA）。[82] Naif Tarafdar、Thomas Lin、Daniel Ly-Ma、Daniel Rozhko、Alberto Leon-Garcia 和 Paul Chow。2019。构建在云中部署现场可编程门阵列的基础设施。数据中心硬件加速器。[83] Ali Tariq、Austin Pahl、Sharat Nimmagadda、Eric Rozner 和 Siddharth Lanka。2020。Sequoia：在无服务器计算中实现服务质量。ACM云计算研讨会（SoCC）。[84] Blaise-Pascal Tine、Sudhakar Yalamanchili 和 Hyesoon Kim。2021。单一源代码硬件—软件协同设计。加速器设计语言、工具与技术研讨会（LATTE）。[85] Diman Zad Tootaghaj、Farshid Farhat、Mohammad Arjomand、Paolo Faraboschi、Mahmut Taylan Kandemir、Anand Sivasubramaniam 和 Chita R. Das。2015。评估节点架构与云工作负载特征对网络流量以及性能/成本的综合影响。IEEE 国际工作负载特征分析研讨会（IISWC）。[86] Liang Wang、Mengyuan Li、Yinqian Zhang、Thomas Ristenpart 和 Michael Swift。2018。窥探无服务器平台幕后。USENIX 年度技术会议（ATC）。[87] Mengzhi Wang、Tara Madhyastha、Ngai Hang Chan、Spiros Papadimitriou 和 Christos Faloutsos。2002。数据挖掘与性能评估相遇：突发流量建模的快速算法。数据工程国际会议（ICDE）。[88] Jagath Weerasinghe、Francois Abel、Christoph Hagleitner 和 Andreas Herkers- dorf。2016。解聚式现场可编程门阵列：与……相比的网络性能

裸金属服务器、虚拟机和 Linux 容器。国际云计算技术与科学会议 *(CloudCom)*。[89] Jagath Weerasinghe、Raphael Polig、FrancoisAbel 和 Christoph Hagleitner。2016。面向数据中心应用的网络连接型 FPGA。国际现场可编程技术会议 *(FPT)*。[90] Yanan Yang、Laiping Zhao、Yiming Li、HuanyuZhang、Jie Li、Mingyang Zhao、XingzhenChen和KeqiuLi。2022。INFless：面向低延迟、高吞吐量推理的原生无服务器系统。国际编程语言和操作系统体系结构支持会议 *(ASPLOS)*。[91] Jianwei Yin、Xingjian Lu、XinkuiZhao、HanweiChen和XueLiu。2015。BURSE：面向云计算的突发性与自相似工作负载生成器。*IEEE*并行与分布式系统汇刊 *(TPDS)*。[92] YueZha 和 Jing Li。2020。云中的现场可编程门阵列虚拟化。国际编程语言和操作系统体系结构支持会议 *(ASPLOS)*。[93] Chengliang Zhang、Minchen Yu、Wei Wang和Feng Yan。2019。MArk：利用云服务实现经济高效且感知 SLO 的机器学习推理服务。*USENIX* 年度技术会议 *(ATC)*。[94] Qian Zhang、Jiyuan Wang、GuoqingHarry Xu 和 Miryung Kim。2022。Het- eroGen：通过自动化测试生成和程序修复将 C转译为异构 HLS 代码。国际编程语言和操作系统体系结构支持会议 *(ASPLOS)*。[95] Shuze Zhao 、 Ibrahim Ahmed 、 Carl Lamoureux 、 Ashraf Lotfi、Vaughn Betz 和 OlivierTrescases。2016。具有热补偿功能的用于现场可编程门阵列节能的通用自校准动态电压与频率调节(DVFS)方案。*IEEE* 应用电力电子会议暨展览会 *(APEC)*。[96] Zhipeng Zhao、Hugo Sadok、Nirav Atre、JamesHoe、Vyas Sekar和 JustineSherry。2020。在单台服务器上实现 100Gbps 入侵防御。*USENIX* 操作系统设计与实现研讨会 *(OSDI)*。