借助数据间接预取器在可重构架构上加速图分析

Yichen Yang、Jingtao Li、Nishil Talati、Subhankar Pal、Siying Feng、Chaitali Chakrabarti、Trevor Mudge、Ronald Dreslinski

摘要——图工作负载的内存访问具有不规则性，导致其在现代计算平台上的性能较差。尤其是在众核可重构架构（MRAs）上，即使是最先进的图预取器也无法良好工作（加速比仅为 3%），因为它们是为传统 CPU 设计的。这是因为 MRA 中的缓存通常不足以容纳大量预取数据，而且许多 MRA 采用共享缓存，而此类预取器根本不支持共享缓存。本文研究了适用于一种名为 Transmuter 的 MRA 的数据预取器设计。该预取器构建于目前性能最佳的 CPU 数据预取器 Prodigy 之上。为使预取器适用于 MRA，关键设计要素包括融合预取器状态处理寄存器和预取握手协议，以支持运行时重配置；此外，还重新设计了 Transmuter 中的缓存结构。对常见图工作负载的评估表明，这些架构的协同集成相比无预取器的基线，平均可实现 1.27× 的加速比，在某些工作负载上最高可达 2.72×。**索引词**——图处理、硬件预取、可重构架构、低功耗设计 ✦

arXiv:2301.12312v1 [cs.AR] 29 Jan 2023

1 引言

随着 Dennard 缩放和摩尔定律走到尽头，CPU 和 GPU 等传统架构已无法满足现代数据中心和高性能计算对性能和功耗的要求。人们提出了领域专用加速器以实现高效率，但其可编程性有限。此前的研究提出了低功耗众核可重构架构（MRA），以弥合可编程性与效率之间的差距。近期的一个例子是 Transmuter（TM） [1], ，它在许多内核和大型端到端应用程序上展现出卓越的性能 [2], [3]。

图分析是一个重要领域，拥有许多现实世界中的应用程序 [4], [5]。这些应用通常涉及不规则内存访问 [6], ，因此在商用硬件平台上的性能受到限制。即使在 TM 这样的最先进（ SoTA）MRA 上运行五种图分析工作负载（第 4 节），平均 L1 缓存未命中率也达到 34%。显然，这表明性能仍有优化空间。

若干先前的研究工作（如 [6], [7] ）提出了软件（SW）和硬件（HW）优化技术，以加速 CPU 上的图分析。其中，数据预取器能够适应图算法复杂的内存访问模式，从而有效降低内存访问延迟。Prodigy [6]是一种当前最先进的预取器，在通用 CPU 上执行不规则工作负载方面展现了迄今为止的最佳性能。在本工作中，我们旨在为众核可重构架构设计并适配一种基于 Prodigy 的预取器，并展示其在常用图工作负载上的性能提升。

我们首先讨论为何将未经改动的 Prodigy 直接纳入 TM 设计只能带来很小的性能收益（根据我们的实验，仅提升 3%）。首要问题在于，Prodigy 会预取大量数据，因此无法直接用于片上缓存相对较小的类 TM 的多核可重构架构；这些数据会挤出热点数据，从而抵消引入预取数据所带来的收益。此外，类 TM 的多核可重构架构所使用的缓存既可以配置为私有模式，也可以配置为共享模式，而 Prodigy 的设计并不支持共享缓存。为使 Prodigy 适用于 TM，我们提出了以下微体系结构改动。在预取器方面，为支持共享缓存，我们提出：1）将每个处理器核心的预取状态处理寄存器（PFHR）融合在一起；2）一种协调预取请求的技术；以及 3）用于取消 PF 请求的适当标准。在 TM 方面，我们重新评估了基本的缓存设计参数，包括 L1 缓存大小以及瓦片与二级缓存存储体的比例，以便更好地容纳预取数据。我们在五个图工作负载上评估了这项工作。结果表明，我们的微体系结构改动使 TM 的 L1 缓存未命中率平均降低了 40%，相较于未采用预取的基线 TM，端到端加速比平均达到 1.27× （最高达到 2.72×）。

总而言之，本文作出以下贡献：1）介绍了将数据间接预取器 Prodigy 适配到 Transmuter（一种最先进的多核可重构架构）时面临的挑战；2）提出了用于将 Prodigy 适配到 TM 的微架构支持，以提升其在图工作负载上的性能；3）进行了广泛的评估，结果表明，增强预取器的 TM 相较于基线 TM 平均实现了 1.27× 的加速比。

2 背景与相关工作

2.1 Transmuter：一种众核可重构架构

Transmuter [1] 是一种高度可重构的众核瓦片架构，同时针对可编程性和效率进行了优化。在最高层级上，TM 由多个集群组成，这些集群连接到高带宽内存（HBM）堆栈。每个集群由多个处理瓦片组成（图 1(a)），并可通过可重构的二级缓存交叉开关层访问 HBM。集群还可以通过容量较小但速度较快的暂存存储器进行通信，该存储器主要用于存储同步变量。每个处理瓦片（图 1(b)）由两类可编程的顺序执行核心组成，这些核心采用浅流水线，分别是本地控制处理器（LCP）和通用处理单元（GPE）。GPE 是工作核心，通过工作/状态队列接口接收来自 LCP 的指令；该接口由一组映射到特殊地址的先进先出缓冲区组成。GPE 通过由可重构数据缓存（R-DCaches）和可重构交叉开关（R-XBars）组成的 L1层与二级缓存连接。本文主要针对该 L1层提出微架构改动。

2.2 Prodigy：面向非规则工作负载的 ASOTA 预取器

Prodigy 是一种软硬件协同设计方案，旨在提升重要的数据间接型不规则工作负载 [6] 的存储器系统性能。为捕捉程序行为，Prodigy 提出了一种称为数据间接图（DIG）的图表示（不同于图算法中的输入数据集）。DIG 是一种加权图，能够有效捕捉关键数据结构的布局和内存访问模式。Prodigy 采用编译器分析技术，自动识别间接内存访问，并将 DIG 表示构建到程序二进制文件中。在运行时，该 DIG 用于对硬件预取器进行编程，使其理解不规则算法的程序行为。每个处理器核心的 L1 缓存中都设有一个 Prodigy 预取（PF）引擎。每个 PF 引擎包含两部分：*DIG* 表 和预取器逻辑（图 1(c)）。预取器逻辑 利用小型存储单元存储 DIG（DIG 表），并利用运行时信息跟踪非阻塞的活动预取序列（PFHR），从而发出符合算法行为的预取请求。它监控来自 CPU 侧的需求请求、来自动态随机存取存储器侧的内存填充，并采用灵活的预取算法

• *Yichen Yang*、*Nishil Talati*、*Subhankar Pal*、*Siying Feng*、*Trevor Mudge* 和 *Ronald Dreslinski* 就职于密歇根大学，安娜堡，*MI 48105*。电子邮箱：*yangych@umich.edu* • *Jingtao Li* 和 *Chaitali Chakrabarti* 就职于亚利桑那州立大学，坦佩，*AZ 85281*。

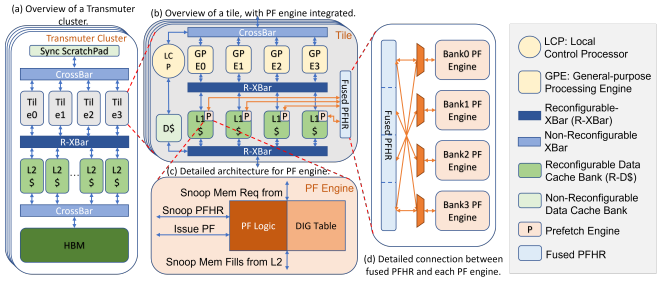

图 1：（a）包含四个瓦片的 TM 集群概览。此层级未引入任何更改。（b）每个瓦片包含四个 GPE 的 Transmuter 瓦片概览。预取引擎连接至每个 L1 R-DCache 存储体。（c）PF 引擎的详细架构。（d）瓦片内 PF 引擎与融合 PFHR 之间的详细连接关系。

该算法通过动态调整预取距离以适应应用程序的执行速度，从而确保预取及时性。本文提出的微体系结构改动主要针对 PFHR 和预取握手协议。

2.3 相关工作

已有工作提出了可在子处理器核心级别 [8] 和网络级别 [9]进行重构的众核可重构架构。Transmuter [1]采用通用核心，并对其片上存储器和互连网络进行重构，从而提供可编程性和灵活性。CoSPARSE [2],与本工作正交，通过在图算法的拉取-推送模式之间进行适配，优化了 TM 上图分析的性能。已有许多关于预取的研究，旨在改善图分析的内存访问延迟。通常，这些方法利用图数据在内存中的布局信息来发出预取请求 [6], [10], [11]。

3 针对 FortranTransmuter 调整 Prodigy

3.1 Prodigy 微体系结构扩展

挑战： 实验表明，图算法通常在共享模式下表现更好（第 5.2.1 节）。然而，Prodigy 最初是针对具有私有 L1 缓存的 CPU 架构设计的。原有的 PF 引擎仅监控预取请求，并将预取数据填充到自身缓存中，而不会与其他 PF 引擎通信。MRA（例如 TM）支持共享一级缓存，甚至提供运行时重配置能力，例如在私有/共享缓存模式之间切换。私有模式类似于一种 CPU 架构：每个缓存存储体中的 PF 引擎仅监控自身流量；而共享模式则有所不同。在共享模式下，数据采用缓存着色技术存储在缓存存储体中，同一瓦片内的所有 PF 引擎必须相互协调，由正确的预取器生成请求，并将数据带入正确的缓存存储体。使用未经修改的 Prodigy 时，我们观察到预取数据被带入了地址未映射到的存储体。因此，预取数据未被使用，并导致缓存污染，相较于无预取基线的性能提升仅为 3%。下面我们将讨论**Prodigy 微体系结构改动**。

3.1.1 融合式 *PFHR* 数组

为了支持 TM 的运行时缓存重新配置，PFHR 需要具备可重配置性。具体而言，由于 PFHR 存储应用程序的运行时信息，因此根据缓存配置，它应由单个 PF 引擎私有使用，或由瓦片内的所有 PF 引擎共享。如图 1(d) 所示，我们提出在一个瓦片内使用**融合 PFHR 阵列**，而不是像原始设计那样为每个 PF 引擎配置一个 PFHR [6]。融合 PFHR 阵列由多个存储体组成。根据缓存模式，PF 引擎会访问融合 PFHR 阵列中相应的存储体，并使用内容寻址存储器（CAM）搜索运行时信息。具体来说，当 L1 缓存为私有缓存时，每个 PF 引擎访问专用的 PFHR 存储体；当 L1 缓存为共享缓存时，所有 PF 引擎在一个瓦片内访问 PFHR 阵列的所有存储体。此外，出于公平性和设计简便性的考虑，每个 PFHR 存储体仅有一个读写端口；在共享模式下，PF 引擎可通过轮询方式访问融合 PFHR 阵列。实验表明，与读写端口数量不受限制的情况相比，这种轮询仲裁对性能的影响可以忽略不计。通过这项微架构增强，在私有缓存模式下，所有 PF 引擎可并行访问各自专用的 PFHR 存储体；而在共享缓存模式下，每个时钟周期只有一个 PF 引擎能够访问整个 PFHR 阵列。

3.1.2 预取请求握手协议

当多个 L1 缓存由多个处理器核心共享时，我们会采用一种类似于页着色 [12], 的缓存着色技术，对 L1 缓存组进行着色，以提高缓存利用率并最大限度地减少存储体争用。然而，在设计中加入预取器后，预取器会生成预取请求，将数据加载到其宿主缓存存储体中；但该存储体可能与根据缓存着色策略映射预取数据、也就是应当从中取回数据的存储体不同。即这违反了缓存着色策略，从而阻止处理器核心访问预取数据。因此，我们提出一种瓦片内各 PF 引擎之间的预取请求握手协议。该技术可确保每个预取请求都由与请求地址所对应的宿主缓存存储体相对应的 PF 引擎发出。根据我们的策略，PF 引擎不会立即发出生成的预取请求，而是遵循缓存着色策略，将预取请求传递给相应缓存存储体的 PF 引擎，并由该预取器发出预取请求。

3.1.3 重新设计的 *PFHR* 清除标准

Prodigy 中的 PFHR 采用清除机制，使预取器始终领先于程序执行。当执行线程追上预取器时，新的运行时信息会覆盖旧信息。然而，当 L1 缓存为共享缓存时，PFHR 数组由所有 PF 引擎共享。由于不同 GPE 核心的进度可能不同，特定 GPE 生成的 PFHR 条目不应被同一瓦片内的其他 GPE 覆盖。为了正确清除 PFHR 条目，我们将 GPE ID 作为附加字段加入 PFHR 条目，并强制规定只有 GPE ID 匹配的条目才能被清除。

3.2 Transmuter 缓存设计变更

挑战： CPU 具有多级大容量缓存（最大可达数兆字节），能够提供足够的存储空间来容纳大量预取数据；而 GPU 以及 TM 等低功耗 MRA 的片上存储则小得多，难以达到相同比例。因此，直接将针对多核 CPU 优化的 Prodigy 集成到 TM 中，会导致较高的 L1 缓存替换率，并造成 L1 到 L2 接口处的拥塞。我们对 TM 中的缓存结构进行了仔细重新设计，以缓解这一 L1 到 L2 瓶颈。具体而言，我们对**TM 架构** 进行了以下更改（详细评估见第 5.2 节）。

3.2.1 调整 *L1* 缓存大小

使用基于Prodigy的预取器会在程序执行前生成大量预取请求，这些请求需要存储在 L1 缓存中以供后续使用。根据我们的实验（第 5.2.2 节），我们发现 L1 的替换次数（将数据装入缓存时驱逐有效块的次数）很高，这表明 L1 缓存太小，无法容纳额外的预取数据。一个直接的解决方案是降低预取器激进程度，但我们发现这会显著削弱性能提升（加速比不到 5%）。我们使用 CACTI 进行的评估表明，在 1 GHz 时钟频率下，容量小于 64 kB 的 SRAM 访问延迟为 1 个周期。考虑到调整大小带来的性能收益，我们决定将每个存储体的 L1 缓存大小从 4 kB 增至 16 kB，为预取数据腾出更多空间，同时略微增加能耗和面积。

表 1：gem5 模型的微体系结构参数。

模块	微体系结构参数
PE/LCP	1.0 GHz 下的单发射、4 级流水线、顺序执行（MinorCPU）处理器核心
L1 RCache（每个存储体）	16 kB、单端口、字粒度、4 路组相联、非一致性 包含 8 个未命中状态处理寄存器、块大小为 64 B、每个 GPE 配备 1 个存储体的缓存
L2 缓存（每个存储体）	4 kB，共享模式，其余与 L1 R-DCache 相同，每个瓦片包含 4 个存储体
主存	1 HBM2 堆栈：16 个 64 位伪通道，每个 @ 8000 MB/s，80-150 ns 平均访问延迟
预取器 PFHR	每个 GPE 8 个条目

3.2.2 二级缓存中的细粒度存储体划分

预取器生成的大量请求也会导致 L1 到 L2 接口拥塞。由于 R-XBars 会串行处理发往同一输出端口的请求，随着预取请求增多，这些请求会滞留在本地缓冲区中等待转发（第 5.2.3 节）。原始 TM 架构将 L2 存储体数量设为与瓦片数量相同，这进一步加剧了这一问题，即 当每个瓦片包含过多 L1 缓存组时，该架构没有考虑到 L2 缓存组所承受的额外压力。我们通过提高 L2 的分组粒度来解决这一问题。具体而言，将每个瓦片的 L2 缓存组数量从 1 个增加到 4 个（即 对于包含四个瓦片的 TM 设计，总计有 16 个 L2 缓存组），同时保持 *L2* 缓存总大小不变。这样可以提升 L1-L2 连接带宽，而不会显著增加面积或能耗开销。

4 方法论

4.1 在 Transmuter 上实现图算法

由于目前没有适用于 TM 的开源图分析实现，我们使用 TM C++ API 为五种算法编写了手工优化代码，包括 PageRank (PR)、PageRank-Nibble (PRN)、广度优先搜索 (BFS)、单源最短路径 (SSSP) 和协同过滤 (CF)。这些实现采用拉取模式，遍历每个节点的入邻接顶点，以判断是否需要自更新，并要求使用压缩稀疏列格式。

4.2 实验设置

我们使用 gem5 [13] 模拟增强型 Prodigy-TM 系统。所用的微体系结构参数与以往工作 [1], [2], [6] 类似，列于表 1。我们重点研究 TM 配置（4 个瓦片，每个瓦片含 16 个通用处理单元），并采用第 3.2 节所述的修改。该设计为每个通用处理单元配备一个 16kB L1 缓存组，并为每个瓦片配备四个 4kB 二级缓存组。L1 和二级缓存采用共享模式（即L1 缓存组共享

在一个瓦片内的 GPE 之间共享；二级缓存组在各瓦片之间共享），除非另有说明。没有预取器的原始 TM 设计作为基线，用于通过我们的五种图应用程序实现评估 Prodigy-预取器的性能。表 2 列出了实验中使用的每个真实世界 [14],[15],均匀随机 [16]和 Kronecker 图 [17] 的详细信息。遵循先前的工作 [1], [2], ，我们使用 CACTI 7.0 [18] 进行能耗估算。我们建立了一个涵盖整个系统的模型，包括重新设计的缓存结构，并通过流片芯片原型 [19]对其进行交叉验证。

5 评估

5.1 总体性能

图 2 展示了在所评估的不同应用程序中，采用我们的 Prodigy-Transmuter 设计后 L1 缓存未命中率的降低幅度和总体加速比。与基线 TM 设计相比，所提出的系统平均实现了 1.27× 倍的性能提升。这些收益直接源于平均 84% 的预取准确率和 40% 的 L1 缓存未命中率降低。因此，我们得出结论：结合 Prodigy 的微体系结构改动与重新设计的 TM 缓存结构，能够带来可观的性能收益。

我们观察到，该系统通常更适合稀疏且分布较均匀的图，例如 cr；对于单源最短路径，cr 的加速比最高（2.72×）。对于这类稀疏图，在缓存大小固定的情况下，与较稠密的图相比，预取器可以比应用程序提前运行更远，从而预取更多数据，更好地隐藏内存访问

表 2：输入图的规格。内存大小根据使用 PR 算法时的内存占用计算。

图	# 顶点	# 边	内存大小	Type
CARoad(cr)	1,965,206	2,766,607	40.540 MB	道路网络
soc-Pokec(pk)	1,632,803	30,622,564	141.730 MB	社交网络
Slashdot0811(sd)	77,360	905,468	4.635 MB	社交网络
ego-Twitter(tt)	81,306	1,768,149	7.986 MB	社交网络
in-2004（in）	1,382,908	16,917,053	85.635 MB	Web网络
Kronecker18（kn）	262,143	3,805,448	18.517 MB	同步图
单向图 1Mx2（um2）	1,000,000	2,000,000	22.888 MB	同步图
单向图 1Mx8(um8)	1,000,000	8,000,000	45.776 MB	同步图

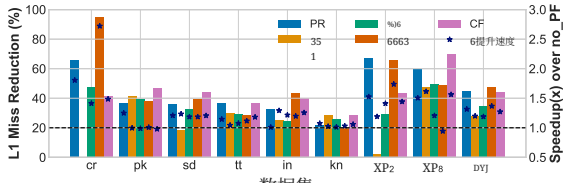

图 2：所提出的系统相较于基线 4×16 TM 实现的 L1 缓存未命中减少（柱形）和加速比（标记）。每项实验均设置了最佳预取器激进程度。CARoad-PRN

由于超出了仿真限制，因此未显示。

延迟。对于非均匀图，当遇到高度连接的节点时，预取器会将其所有邻居数据载入缓存，从而逐出先前已预取、即将使用的数据。因此，这些被逐出的数据的内存访问延迟会增加，进而影响整体性能。

我们强调，我们报告的改进是相对于 TM 基线而言的；对于稀疏代数和图分析领域，该基线相较于商用硬件已经实现了更高的性能提升 [1], [2]。因此，在相同工作负载下，我们的方案预计可实现相对于 CPU（运行 MKL 库的 i7-6700K）的 4.6× 倍性能提升，并配合 4×16 TM。

由于篇幅限制，我们仅对 Prodigy 与其他预取器进行了定性比较。Prodigy 能够捕获图分析中普遍存在的两种通用数据访问模式，并能够根据应用程序的进展发出预取请求 [6]。对于不规则的内存访问模式，步长预取器和相关预取器等较为简单的预取器由于不了解应用程序的运行时信息，因此效果有限。

5.2 设计空间探索

我们进行了全面的设计空间探索，以选择预取器增强型 TM 配置的最优方案。

5.2.1 私有 *L1* 缓存与共享 *L1* 缓存

尽管数据相关内存访问是图算法中的主要模式，但时间局部性仍然存在，尤其是在幂律图中。度较高的节点往往会被更频繁地重新访问。共享缓存比私有缓存更能利用这种局部性。在无预取器的情况下运行 PR 时，共享 L1 系统的平均性能比私有 L1 系统高 1.51×，对于 Slashdot 数据集，加速比最高可达 2.68× 。即使在两种模式下都启用预取器，共享模式的性能仍比私有模式高 1.33× 。因此，共享 L1 缓存通常具有更好的性能。

5.2.2 L1 缓存大小

图 3（左）展示了在有无预取（PF）情况下，不同 L1 缓存大小带来的性能提升。PF 与

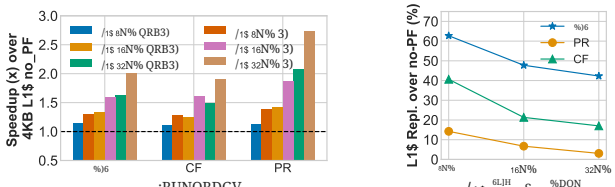

图 3：L1$ 大小实验。左：对于不同的 L1$ 大小，在每个存储体使用 4KB L1$ 且无 PF 的 TM 基础上，启用和不启用 PF 时的加速比。右：相对于无 PF 配置，额外 L1 缓存替换的百分比。所使用的数据集：cr、pk、sd、tt、in、um2 和 um8。

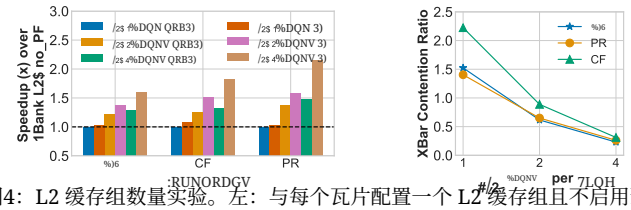

图4：L2 缓存组数量实验。左：与每个瓦片配置一个 L2 缓存组且不启用预取相比，在每个瓦片配置不同数量的 L2 缓存组、启用和不启用预取时的加速比。右：启用预取时，每个瓦片配置不同数量的 L2 缓存组对应的 L1至 L2 R-XBar 竞争比。使用的数据集：cr、sd、tt、um2 和 um8。

较大的 L1 缓存搭配时效果更好，因为系统可以容纳更多预取数据，从而有效隐藏内存访问延迟。在本例中，其优势趋于饱和（每个存储体达到 32kB 后）。我们还从图 3（右）中观察到，L1 缓存的替换次数远高于无 PF 配置，并且随着 L1 缓存容量增加而有所改善。由于缓存访问延迟仍为 1 个周期，增大 L1 缓存大小可提升总体性能（最高至 32kB）。与 4kB-noPF 设计相比，16kB-PF 配置的面积增加了 20%（原始 64×64 TM 设计仅占 CPU（i7-6700K）面积的 1.7% [1]），能耗增加了 22%。不过，该系统可实现 1.68× 的加速比。与 4kB-noPF 相比，按能耗延迟积（EDP）衡量，4kB-PF、8kB-PF、16kB-PF 和 32kB-PF 设计分别降低了 +2%、+15%、 +22% 和 -2%。因此，我们选择每个存储体配备 16kB L1 缓存的设计，以面积和能耗换取更佳性能。

5.2.3 二级缓存组数量

使用基于Prodigy的预取器会使 CPU 侧生成更多内存请求。我们在保持二级缓存总大小不变的情况下，测试了每个瓦片配置不同数量的 L2 缓存组。图4（左）展示了增加 L2 缓存组数量和启用预取器带来的性能提升；图4（右）展示了启用预取器时 L1至 L2 R-XBar 的竞争比。竞争比由模拟器统计：在若干周期内，将 R-XBar 中排队的数据包总数除以通过 R-XBar 的数据包总数，再对整个工作负载执行过程取平均值。竞争比越高，表示输入端口因请求而越拥堵。每个瓦片只有一个 L2 缓存组时，内存请求会受 R-XBar 处理速度限制（竞争比更高）。增加 L2 缓存组后，R-XBar 每个周期可以处理更多请求，从而提高 L1-L2带宽，改善性能。增加 L2 缓存组还会降低二级缓存发生存储体冲突的概率，进一步降低竞争比。

然而，增加 L2 缓存组的数量会导致每个瓦片配置两个存储体后出现性能饱和。只有启用预取器，系统才能充分利用更大的缓存带宽。因此，我们将 4×16 TM 的 L2 缓存配置为每个瓦片 4 个存储体，总共 16 个存储体。

5.3 开销分析

5.3.1 面积和能耗开销

采用先前工作的模型， [6], Prodigy 在面积方面为每个 GPE 引入了可忽略不计的 0.28kB 存储开销（PFHR 和 DIG 存储）。Prodigy 的能耗开销（PFHR 和 DIG 存储）通过使用块大小和条目数量与 PFHR/DIG 相匹配的全相联缓存进行估算。平均而言，预取器的开销仅占系统总能耗的 3.42%，可以忽略不计。如果将 L1 大小的增加视为面积开销，则最终设计相比原始的 4kB-L1 Transmuter 增加了 20%。

5.3.2 不同配置下的性能与能效图 5 展示了不同 TM 配置的加速比和能效提升。实验中，L1 和 L2 缓存的总大小保持不变。工作负载的性能

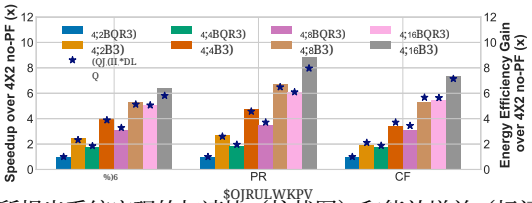

图 5：所提出系统实现的加速比（柱状图）和能效增益（标记）。数据已根据无预取器的 4×2 TM 的执行时间进行归一化。

随着 TM 规模的增大而良好提升。我们还观察到，在时间和能效方面，配备预取器的较小 TM 比未配备预取器的较大 TM 性能更好（平均加速比为 1.15× ）。因此，我们得出结论：与未配备预取器的较大架构相比，配备 Prodigy 预取器的较小架构在图分析方面效率高得多。

6 结论

本文介绍了为 Transmuter（TM）众核可重构架构（MRA）设计的不规则工作负载预取器。我们阐述了为何直接引入已有预取器不足以提升 MRA 的性能，而该预取器原本是为多核 CPU 系统设计的。我们介绍了在搭载 Prodigy 的 TM 中所做的一系列微体系结构改动，使其能够配合 MRA 的运行时缓存重配置。我们还重新设计了 TM 的缓存结构，以存储更多预取数据，并缓解 L1 到 L2 接口处的拥塞。仿真结果表明，与不带预取器的基线系统相比，所提出的系统平均加速比为 1.27× 。受篇幅限制，本文仅展示了我们在 TM 上的方法。不过，该方法可轻松扩展，用于将通用预取器适配到 MRA 上。

参考文献

[1] S. Pal、S. Feng 等，“Transmuter：通过存储器与数据流重配置弥合效率差距”，载于 PACT 2020。[2] S. Feng 等，“CoSPARSE：面向图分析的软件与硬件可重构稀疏矩阵向量乘框架”，载于 DAC 2021。[3] S. Pal、J. Beaumont 等，“Outerspace：一种基于外积的稀疏矩阵乘法加速器”，载于 HPCA 2018。[4] A. Mislove 等，“在线社交网络的测量与分析”，载于 IMC 2007。[5] A. V. Goldberg 和 C. Harrelson，“计算最短路径：搜索遇上图论”，载于 SODA，2005。[6] N. Talati 等，“Prodigy：通过软硬件协同设计降低数据间接型不规则工作负载的存储器延迟”，载于 2021 HPCA。[7] S. Beamer 等，“GAP 基准测试套件”，CoRR，2015。[8] V. Dadu 等，“通过利用常见的数据依赖形式实现通用加速”，载于第 52 届 MICRO，2019。[9] Y. S. Shao 等，“Simba：采用多芯片模块架构扩展深度学习推理”，载于第 52 届 MICRO，2019。[10] S. Ainsworth 和 T. M. Jones，“利用数据结构知识进行图预取”，载于 2016 ICS。[11] A. Basak 等，“图处理工作负载的存储器层次结构分析与优化”，载于 2019 HPCA。[12] G. Taylor 等，“TLB 切片：一种低成本、高速的地址转换机制”，载于 ISCA，1990。[13] J. Lowe-Power 等，“gem5 模拟器：版本 20.0+”，CoRR，2020。[14] J. Leskovec 和 A. Krevl，“SNAP 数据集：斯坦福大学大型网络数据集合集”，2014。[15] T. A. Davis 和 Y. Hu，“佛罗里达大学稀疏矩阵集合”，《ACM 数学软件汇刊》，2011。[16] A. A. Hagberg 等，“使用 NetworkX 探索网络结构、动态特性与功能”，载于第七届 Python 科学会议论文集，2008。[17] J. Leskovec 等，“Kronecker 图：一种网络建模方法”，《机器学习研究杂志》，2010。[18] R. Balasubramonian 等，“CACTI 7：用于新型片外存储器互连探索的新工具”，《ACM 架构代码优化汇刊》，2017。[19] S. Kim 等，“Versa：一种具有动态可重构互连与存储器的 36 核脉动式多处理器”，《IEEE 固态电路期刊》，2022。