# 利用机器学习最大化硬件预取效率

# 文献翻译

计算机学院

13070038 李芳达

# 利用机器学习最大化硬件预取效率

Saami Rahman, Martin Burtschery, Ziliang Zongz, and Apan Qasemx

Department of Computer Science

Texas State University

San Marcos, TX 78666

saami.rahman@txstate.edu, y burtscher@txstate.edu, z ziliang@txstate.edu, x [apan@txstate.edu](mailto:apan@txstate.edu)

***摘要***——**现代处理器配有多个硬件预取器，每个预取器针对特定的存储层次，并且使用各自独立的预取算法。但是，为了使不同程序的运行性能达到最大，需要采用不同的预取器子集。开启所有可用的预取器很难产生最佳性能，并且在某种情况下，预取甚至会降低性能。**

**这篇论文讨论了多线程代码的硬件预取效果，并展示了一种使用机器学习****预测出给定应用程序的最佳预取器组合的技术。这项技术基于程序的特性化并利用硬件性能事件结合剪枝算法来获得简明可表达的特性集。得出的特性集被用在三种不同的学习模式中。所有必要步骤都在一个框架内执行，能达到最大值的平均96%的预取加速比。本框架使用开源工具建立，因此易于扩展和移植到其他硬件架构。**

1. 介绍

预取监视器和外推流获取应用程序的模式然后将数据超前预取到更高层次的缓存中来降低内存延迟。由于CPU和内存之间的速度差，预取在代码优化中一直是一个重要的技术。在现在的多核结构中预取变得更加具有重要意义。随着被附加在同一个存储系统上的核心数量的增加，通常来自同一程序的不同线程中的预取候补流数量也会增加。有效的多预取流协调不仅能降低存储延迟，还能直接影响并行效率。

虽然预取在现在的体系结构中是一种关键的转型，决定优化预取参数造成了多个挑战。第一，预取器必须精确地预测存取模式。如果预测错误，就会增加存储访问负担，并且更重要的是，会在空间很小且贵重的缓存中造成冲突。第二，预取指令必须及时。如果预取造成数据早于需要之前被放置到更高层缓存中，可能会被那些更紧迫需要的数据覆盖掉。这些挑战在多线程程序中被更进一步地放大。L2等更低层次的内存可以被多个线程共享，每个线程可能需要不同位置的数据，准确地决定出读取顺序是一件困难的事情。

在这篇论文中，我们会处理这些挑战，并制定出了一种基于机器学习技术的有效的预取策略。我们探讨在PARSEC程序

[[[1]](#endnote-1)]和两个理想化的顺序和随机存取模式程序上的预取效果。我们在授权给独立硬件预取器，然后将它组合到一个内置4个预取器的英特尔处理器上，以检测性能。为了得到所给程序需要使用的预取器的建议，我们尝试了数个机器学习算法。

一个成功使用基于机器学习的方法的关键步骤是将程序以定量的方式特性化，以获取程序的基本区别。我们使用硬件性能事件来达到这个目的。由于有数百种可用的时间，我们设计了一种简单但是有效的程序来减少特性化一个程序需要的事件的数量。我们通过测试决策树上的不同特性集来展示剪枝算法的效果。最后，我们开发了一个为未知程序推荐硬件预取配置的框架。这个框架可以使用户获得硬件预取器提供的平均为96%的加速比。

使用这种推荐系统调整硬件配置有几个优点。第一，可以使我们更有效地利用已有硬件。第二，无需更改被优化程序的源代码。即可直接适用于可执行程序。最后，框架依赖于开源技术，因此易于扩展和移植到其他体系结构。

1. 相关工作

预取是被广泛开发过的领域，预取的益处被广泛地记录和研究过。Lee et al.[[[2]](#endnote-2)]是第一批提出数据预取概念的研究者之一，通过硬件的数据预取来减少存储访问延迟。Chen et al.[[[3]](#endnote-3)]在他的早期工作中，研究了硬件和软件数据预取的效果。之后人们在数据预取上进行了许多研究工作。

随着多核处理器的提出，预取的效果变得更加有趣。预取会降低性能，这种情况造成的不良后果在[[[4]](#endnote-4)]，[[[5]](#endnote-5)]中已经探讨过。Puzak et al[[[6]](#endnote-6)]论证了预取在何处降低了性能并指出了一种基于时间线的特性化预取，这种方法覆盖范围广且准确度高。Lee et al[[[7]](#endnote-7)]引导了一种深入的关于何时以及为何预取工作的调查。他们对软件和硬件预取在SPEC CPU2006标准程序上的串行工作负载性能进行了广泛地分析。

Jayasena et al[[[8]](#endnote-8)]论证了一种决策树的效果，减少了检测false sharing使用的性能事件数量。他们的方法激发了我们将决策树纳入我们的工作之中。Cavazos et al[[[9]](#endnote-9)]开发了一种机器学习模式，能够找到SPEC CPU2006标准程序组的最佳优化配置。他们工作的目标是找出一组最优化编译，使程序得到性能提升。他们报告了使用性能事件作为学习算法特征时的高分类精准度。Milepost GCC是一个针对更进一步的多源优化学习的大项目。Milepost GCC[[[10]](#endnote-10)]使用了静态特征来描述程序的特征。Demme et al[[[11]](#endnote-11)]做了类似的工作使用基于程序数据和控制流的图表聚类，来描述程序的行为。然而，由于他们修改了编译器或程序代码的中间表示，这种描述很难移植到其他平台。

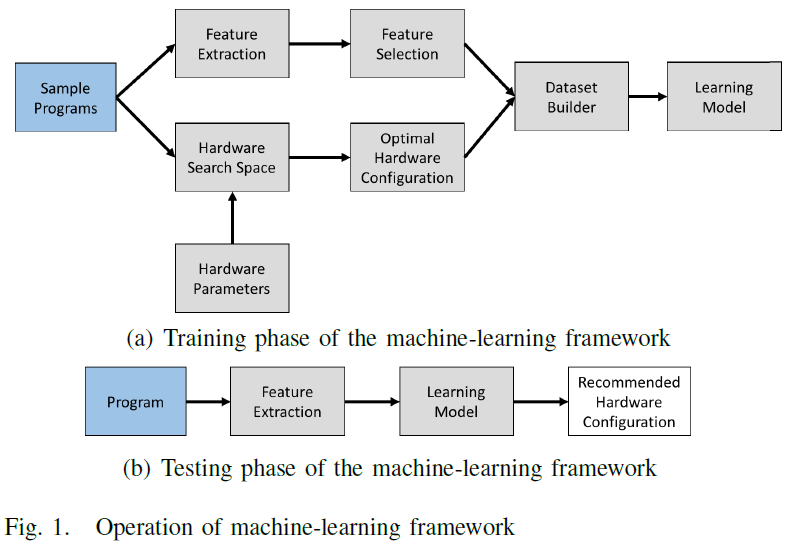
在我们讨论的这些工作中，以下两个与我们的工作最为接近。McCurdy et al[[[12]](#endnote-12)]使用性能事件描述了预取对系统应用程序的影响。他们试验了AMD处理器上的多种标准程序的组合。他们的工作主要集中在串行工作负载，但他们也讨论了同时运行串行程序的效果。他们的工作取决于成功地使性能事件孤立，这足以表现预取效果的获取。他们精选了试验中AMD体系结构的性能事件，并会在任何新体系结构上重复研究所有可用性能事件的步骤。Liao et al[[[13]](#endnote-13)]展示了一种基于机器学习的方法，用于选择优化预取配置，这种方法与我们的方法相似。然而，他们的工作也取决于制定了特定的体系结构性能事件，并且是手动制作的。另外，他们的工作集中在串行工作负载，而我们对并行程序更感兴趣。

1. 机器学习框架

图片1展示了我们机器学习框架要执行的主要任务。共有两个关阶段：训练和测试。在训练阶段，运行样本程序用来生成特征。此外，程序在所有可能的预取器设置下运行，以确定性能并确定优化硬件配置。这个信息用来训练学习模块。测试阶段很简单。第一步，将未使用过的程序运行一次以收集其特征。第二步，将这些特征送入训练模块，得出推荐的预取配置。

1. *预取器配置*

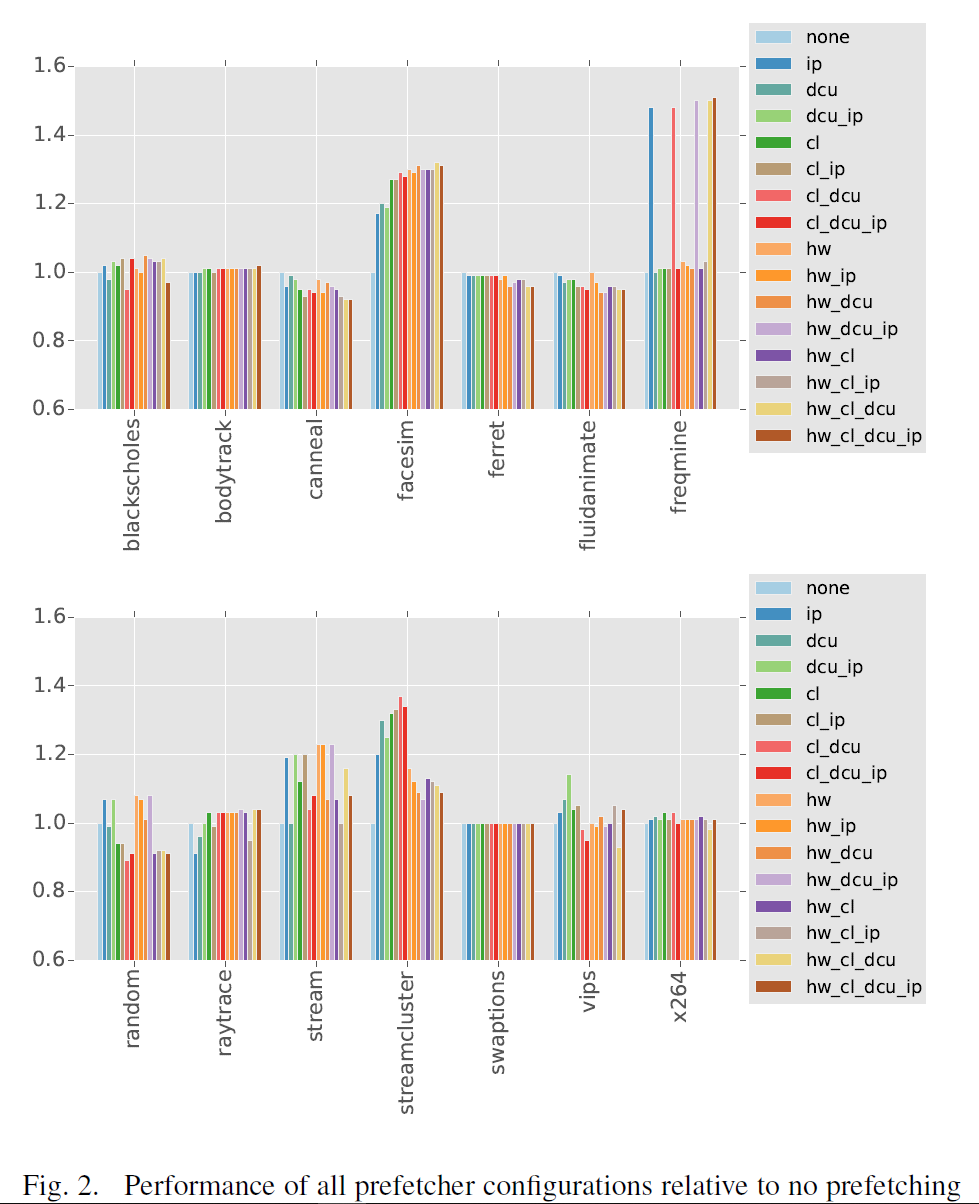
许多现代处理器使用硬件预取器。例如我们使用英特尔酷睿2处理器，它带有如下四个预取器：



1. **Data cache unit (DCU) prefetcher：**此预取器会识别流算法并预取下一行到L1 data cache。
2. **Instruction pointer (IP) based stride prefetcher：**此预取器只会跟踪load指令，并检测跨步访问。它可以检测最高2kB的步长并且预取到L1 cache。
3. **Spatial prefetcher (CL)：**此预取器通过读取成对的数据行将一个cache line放入L2，以组成一个128Byte对齐的区块。
4. **Stream prefetcher (HW)：**此预取器检测L1 chache的流请求并将预期的chache lines放入L2和LLC。

需要注意的是最近更多的英特尔架构，例如SandyBridge 和 Haswell，都包含了同样的预取器。有趣的是，缺省条件下这四个预取器并没有被定义开启或关闭。在本工作中，我们定义了一种配置，用四位掩码的形式表示预取器的开启/关闭状态。掩码的每一位代表一个预取器。1值意味着对应的预取器开启，0值意味着关闭。掩码代表的预取器按照他们的重要程度的降序排列：HW，CL，DCU，和IP。我们使用开源工具Likwid控制处理器的预取器配置[[[14]](#endnote-14)]。

如果训练所有的预取器，例如配置1111，似乎能得到最佳的性能结果。然而，其他配置能够表现的更加优秀。事实上，图片2中说明了在某种情况下，配置1111反而降低性能。报告中的加速比值都是从我们的基准配置0000实时改进的。对每个配置，我们运行三次然后使用最佳运行时从操作系统的抖动中保护结果。因为运行产生的最佳性能不仅有最少的终端，还能在最短时间内运行完成了任务，我们在努力提高结果以接近最佳值。



串行的观察可以通过图片2完成。第一，在同一程序中，改变预取器的配置对性能有着重要的影响。例如，freqmine和streamcluster在调整配置后性能上有着很大差异。第二，配置1111并不总是最佳选项，可参考random, stream, streamcluster, 和 vips的情况。第三，配置0000可能成为最佳配置，也就是说使用预取器反而会降低性能，参考canneal, fluidanimate以及串行配置的random的情况。最后，在串行情况下，有多种“好”的配置，也就是说串行配置的表现与最佳表现相差无几。

1. *特征提取*

我们测量所有受我们使用的处理器支持的性能事件，并使用它们作为描述程序的初始特征集。计数事件例如L2 cache丢失率和停滞CPU周期证明一个程序行为的可数灵视，并且被抢先使用在描述程序中。然而，使用所有的可用事件是有问题的，因为这要消耗很长的测量时间，而且事件会交叉使用处理器。我们使用下面描述的剪枝算法来解决这些问题。

特征一旦生成，会正规划为事件数/每百万执行指令。这样的缩放有必要用来比较不同程序和同一程序的不同运行情况。例如，对一个执行十亿条指令的程序来说一百个页错误也许并不严重，但对只执行一万条指令的程序来说非常严重。对于执行一万条指令的程序来说，正规化后的数值会比十亿条指令的程序得到的值更高，因此能够正确地获取严重程度的差别。

图片3展示了我们使用的14个程序的所有事件的计数。X轴代表事件，轴被移除意味着这张图只是用于形象化表示。两个关键观察点可以从图中找到。第一，许多事件有相近的值，因此他们对程序行为并没有造成差别的信息。第二，许多事件有着很高的值，并使其他有更小域值范围的事件消失。我们通过特征缩放来解决这个问题。

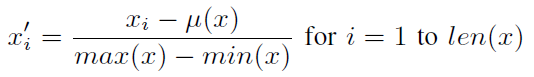
1. *选择特征*

我们使用算法1来减少特征集。算法1由两个子程序组成：FindEvents 和 EventsUnion。FindEvents获取两个程序的事件计数作为输入。对于每个特定的事件，此子程序检查两个数值是否至少相差Φ，Φ是以范围在（0.0,1.0）的数字表示的相对差。如果是，事件包含在集合中。在子程序的最后，集合包含了所有充分获取两个程序区别性的事件。Diff函数返回它的参数除以他们的最大值后的绝对差值。我们试验发现Φ=0.95非常合适，并且只使用这一阈值。

EventsUnion子程序将一列程序作为输入，然后对所有可能的程序对调用一次FindEvents子程序。它包含了多个返回集，最终包含了所有程序对的表现的事件。接着，EventsUnion子程序使用一个map计算每个事件在这些返回集中出现的次数的数值。最后，map中存储了所有数值。

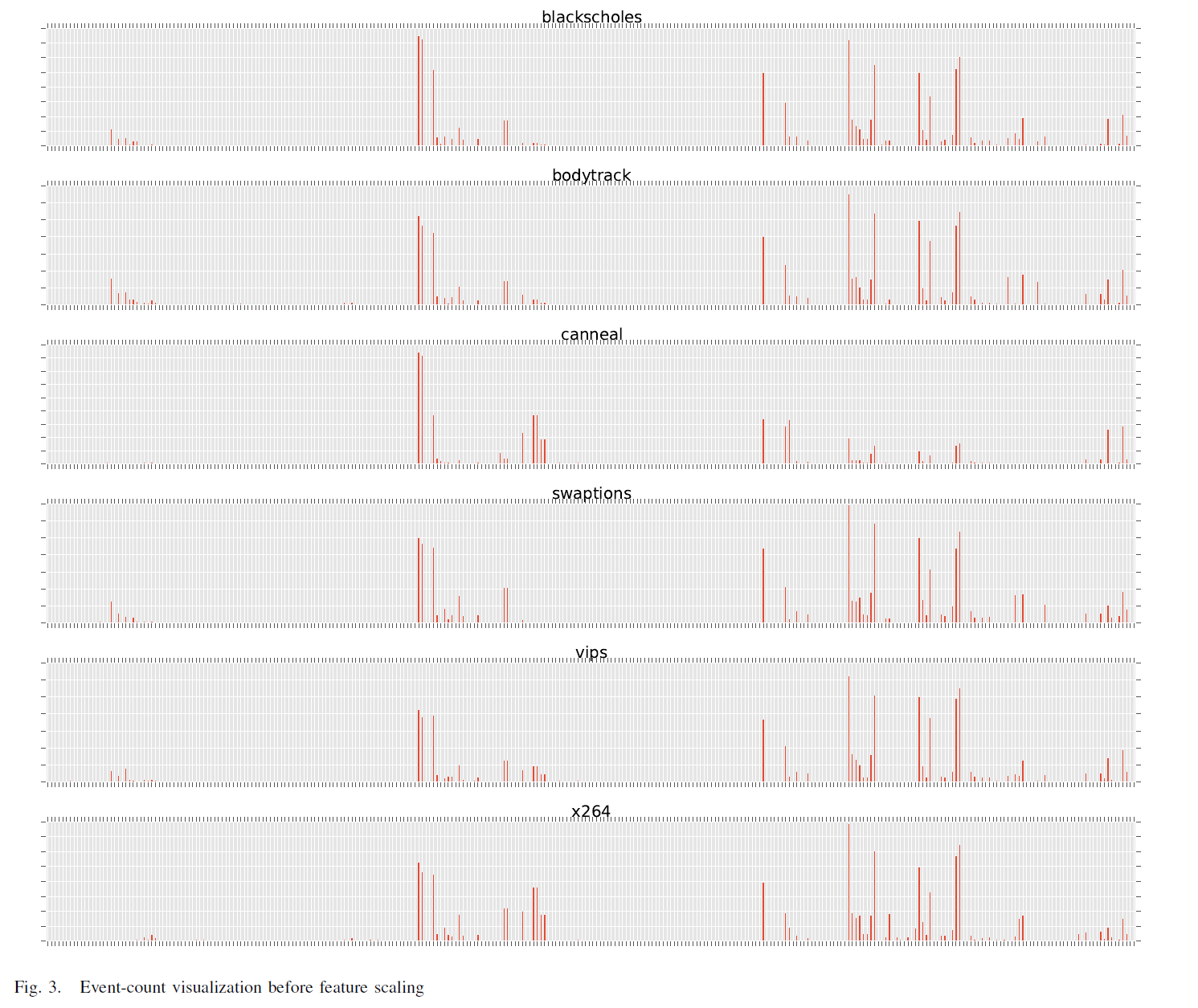
这个算法的思想是，如果一个事件在两个程序中有着很高的特定数值，它获取了两个程序差别的重要形势，因此我们要考虑选择这个事件。然而，很多事件对任何两个程序来说都能明显显出差别，我们最后会得到大量的事件，因此我们使用map给事件排优先级，map中表明了事件在区别两个程序时起到作用的频数。排序后的map我们可以简单地选取最重要的事件来建立我们的最终特征集。

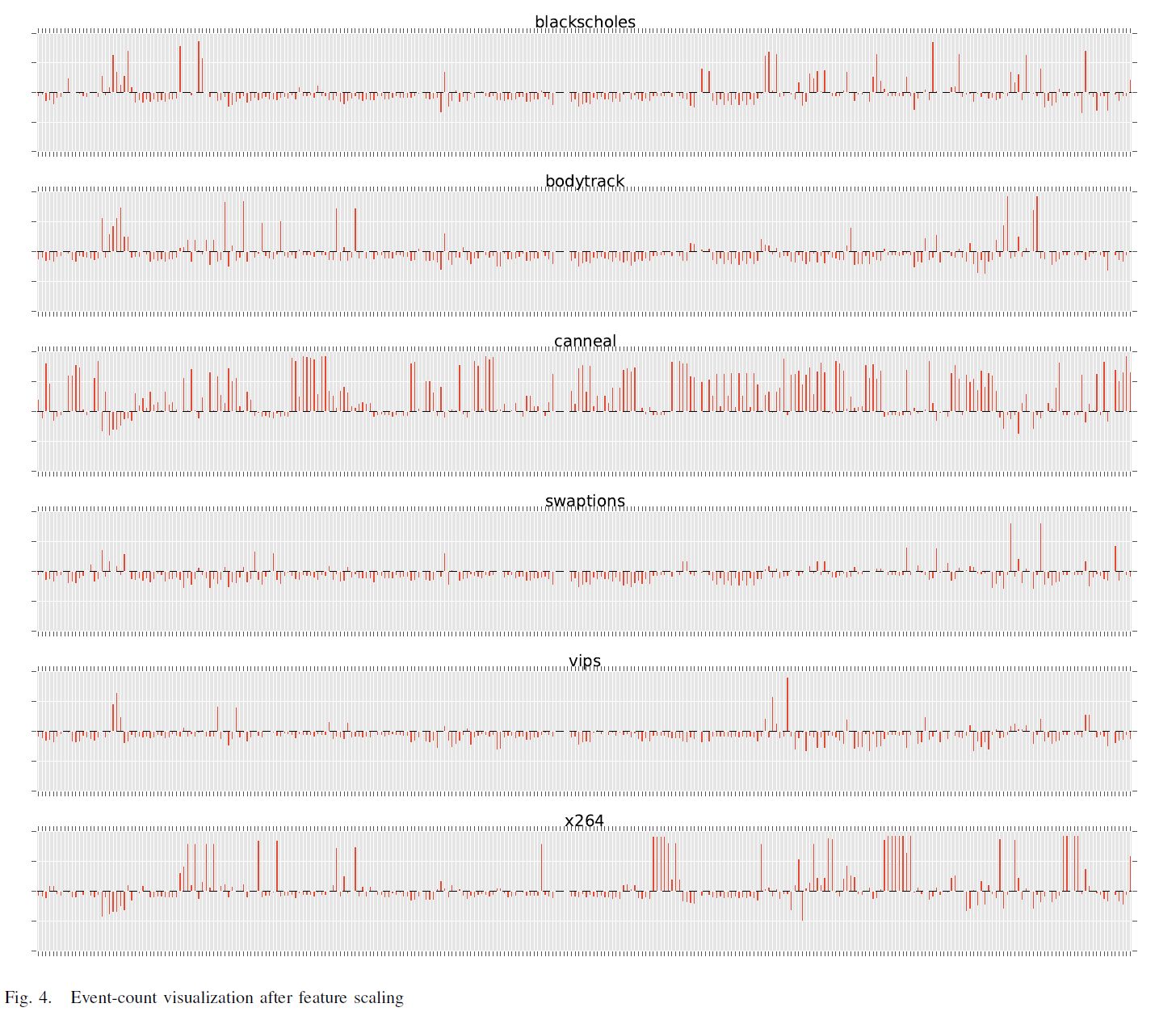
学习阶段前的最后一步，我们称为特征缩放。学习模块用于当特征的数值范围有很大分歧之时。例如，如果一个特征的范围是1~10而同一个特征集里的另一个特征的范围是-10，000~+10,000，学习模块可能画出精确的决定边界。作为补救，我们将所有特征通过下列公式缩放为向量x：



μ(x)是特征向量x中值得平均值，进行缩放后，所有特征的范围会在-1~+1之间。

图片4展示了特征缩放的效果。绘图与图片3有着明显的不同。并且，程序行为的差别也更加明显。例如，缩放前blackscholes和bodytrack的图表比缩放后的图表更难辨别。

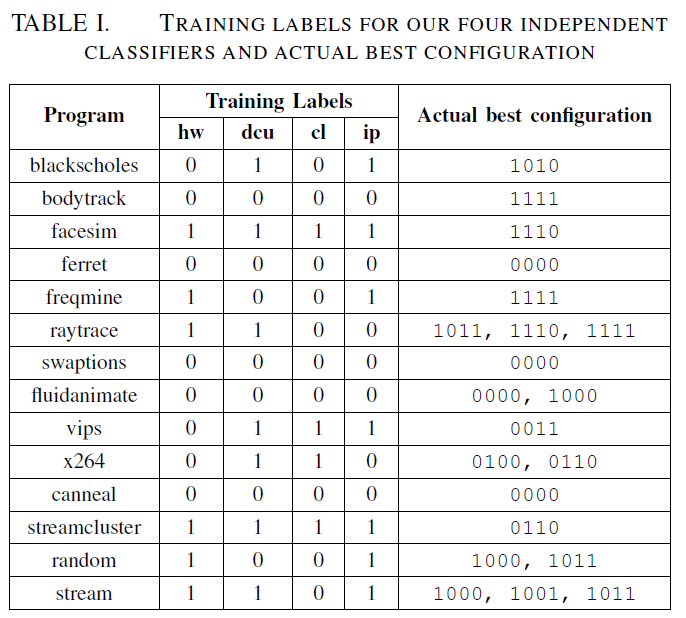
canneal缩放后大体上表现得和blackscholes、bodytrack有了区别。这是因为许多事件数量在canneal中并非0，与图片3中的表现相反。类似地，swaptions, vips, 和 x264在没有缩放时看起来十分相似，但在特征进行缩放后已经有明显的差别。对其他程序也是同样的结果。未缩放的特征的图表稀疏，相反缩放后不同程序的特征图表大体上差别更大，而且密集。



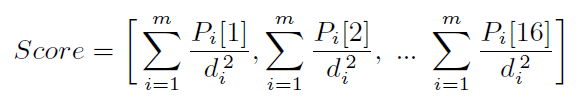
1. *学习模块*

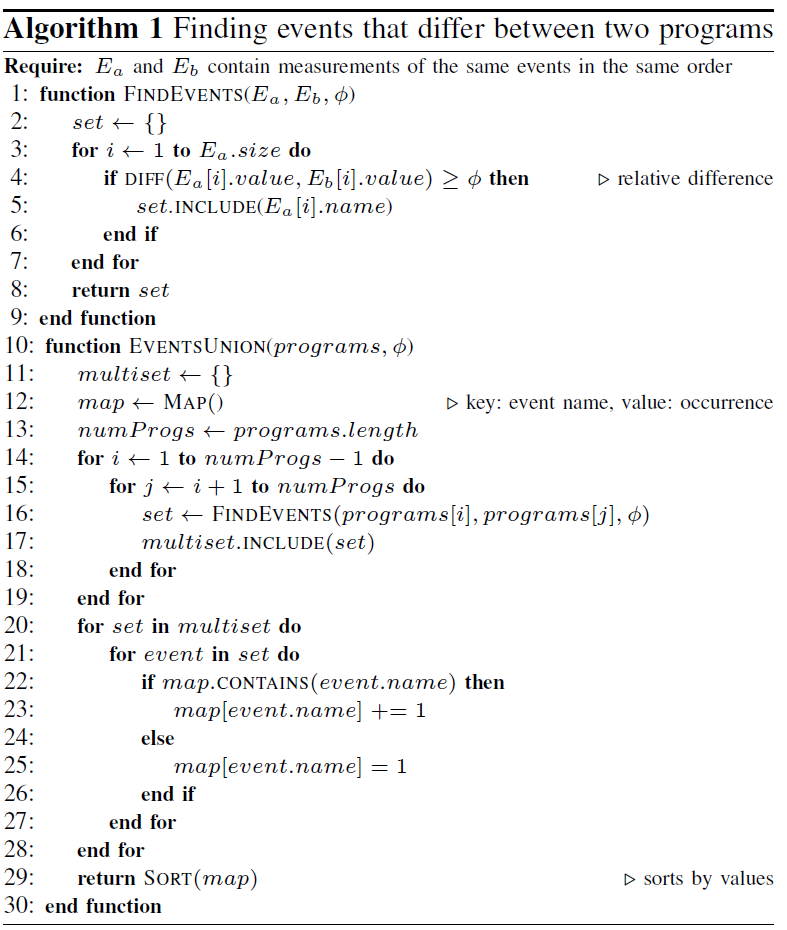
我们的框架的最后一部分是学习模块。我们评估两个不同的传统学习模块，逻辑回归和决策树，并设计了一种基于欧几里得几何的分类器，剪裁到我们需要的并且获取有用的信息。

1. *训练标签：*为了达成两种目的，学习目标将以两种方法公式化。在第一个公式中，我们使用一个二进制分类器来制定程序是否从预取中获益。如果最佳预取配置相对配置0000的加速比大于等于10%，则认为程序从预取中获益。根据这个矩阵，程序ferret, swaptions, fluidanimate, 和 canneal并不获益。我们用这个公式决定要使用哪些算法1中得出的顶层事件。第二个公式单独检查每个预取器的效果。如果为给定程序开启一个特定预取器，使得其相对于配置0000的加速比大于等于2%，我们将这个预取器归类为对此程序有用。这里执行四个学习模块的实例。它同时列出了每个程序的最佳配置。需要注意学习模块对每个类型标签，都应该被多个样本训练。因此要使用四个实例而非一个包含所有预取器组合的实例，等等。16个特定类别标签，需要比我们的14个程序更大量的训练程序。



1. *基于欧几里得距离的模块：*我们使用欧几里得距离作为预测未知程序的好预取器配置时的相似矩阵。这个方法背后的动机是四个独立的分类器不会获取预取器之间的相互作用。将分类器与表1中的最佳配置比较说明了这个问题。分类器的组合通常不是最大加速比的配置。例如，程序raytrace的个别推荐预取器是hw和dcu，对应的配置是1100。然而，这两个预取器的组合并不表现出最佳性能。为了抵消这个问题，我们的欧几里得模块从训练集中计算未知程序和所有已知程序的距离。然后用这个举例作为权重，计算出每个16种可能的配置的得分。具体来说，模块计算下列16元向量：





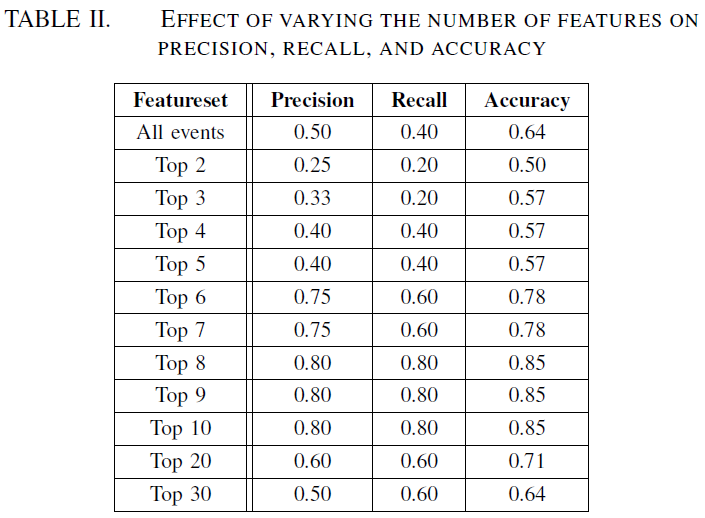
m是训练集中程序的数量。Pi是一个与程序i关联的16元向量。此向量中的每一元对应一个配置，元的脚标的值对应相同二进制值得配置。元的值代表了在程序i上使用对应配置后与可达的加速比的分数。例如，P2[3]代表第二个训练程序使用配置0011获得的加速比分数，3(10)=0011(2)。最后，di指示了未知程序和第i个训练程序的欧几里得距离。

结果向量中包含每个预取器配置的权重分数。推荐配置是归属于最大元的索引的二进制表示。这个方法结合了每个预取器配置对每个训练程序的效果，并使权重得分使用距离的平方。因此，相似程序的预取性能比不同程序带有更大的权重。此外，如果未知程序和数个程序相近，我们的方法可避免最相似的程序成为唯一的推荐基础。

4．结果与分析

*A．实验环境*

我们在2.4GHz英特尔酷睿2 Quad Q6600处理器上执行测量，处理器带有8个32KB L1 caches和两个4MB L2 caches。所有程序使用GCC 4.8.2下-O2优化级别编译，操作系统为Ubuntu 14.04。PARSEC程序由parsecmgmt脚本调用，装入程序集中并用八个线程的native输入。我们在不同的线程下重复试验，但结果并没有显示出预取器的效率有明显的改变。



*B．评估方法*

我们使用(k-1)交叉验证评价我们的学习模块。起初，我们集中在预测准确度。然而并没有获得推荐的质量，因为多预取器配置会产生近似最佳性能。例如，streamcluster的最佳配置是0110有着1.37的加速比，但0111的结果是1.34。因此我们发现哪个可达加速比的分数可以通过学习模块的推荐获得时更加有用。

1. *选择特征最佳数量*

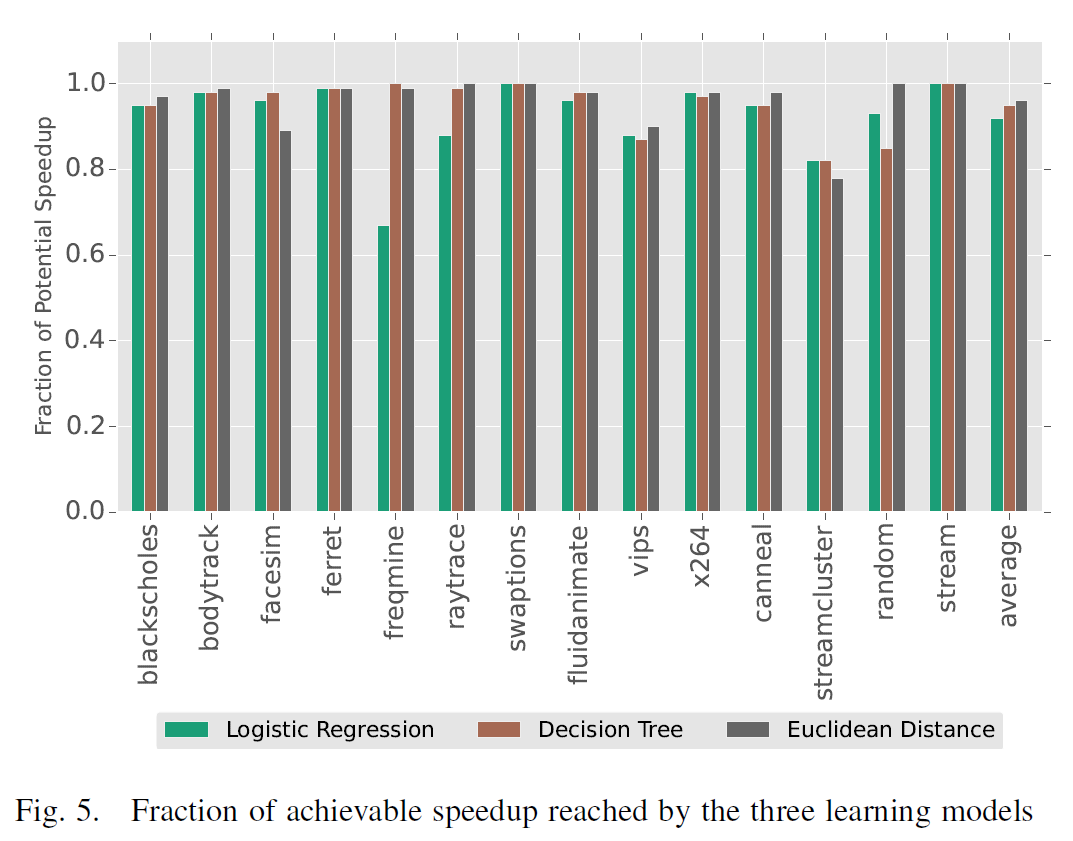
为了测试算法1中选择特征程序的功效，我们用最高的两个事件训练并测试了决策树，然后和数量增加的事件一起送至模块，以此来确定保和点。参考表2此点在我们使用最高的八个事件时达到最大。大于八个事件会降低学习性能。这个剪枝算法的示例可以指定真正获取程序区别的事件。此后，我们使用最高的八个事件作为特征集，唯一的例外是欧几里得模块，我们发现只在最高六个事件下效果最佳。

1. *推荐好的预取配置*

图5展示了三种学习模型的性能下，推荐配置和最佳配置加速比的接近程度。

逻辑回归模型达到了平均92.4%的准确度。这种模型在freqmine上的表现很差，原因是训练标签的派生方式。分别接受cl和dcu预取器对此程序的没有获得加速比，既模型被训练关闭它们的原因。然而，对freqmine程序的好的配置中这两个预取器都启动了。因此，这是两个预取器在单独情况下不提升加速比，但组合的加速比要比他们单独的和更高的一种情况。

决策树模型比逻辑回归模型有着更好的表现，达到了平均95.3%的准确度。对于freqmine程序，可以得出最佳配置的结果。然而，由于预取器之间的相互作用没有被加入到独立模型中，这种情况只会偶然发生。



欧几里得距离模型达到了平均96.1%的准确度。这个模型可以为freqmine程序推荐出好的配置，因为模型考虑到测试程序和训练程序组之间的相似性。此外，模型利用了所有训练程序中每一个可能的预取器配置的性能信息。相比之下，其他分类器基于独立的预取器影响来预测最佳配置。

欧几里得距离模型在facesim程序的表现上大致低于其他模型。因为它认为random是最为相似的程序。Random的高效配置极大地影响了模型的决定。当我们检查模型内部，我们发现第二顺位的推荐是0110，达到了98%的可达加速比。很明显。相似矩阵在这种情况下显示出了不完美性。

所有模型在对streamcluster的运行结果都很差。逻辑回归和决策树分类器由于受到上述问题的影响，虽然训练标志是1111但实际的最佳配置是0110。欧几里得距离模型结果不佳是因为streamcluster与其他任何程序相比都有明显差异。因此在训练集中没有相似的程序。此程序和stream最接近，使用stream的配置可以使加速比达到1.07，相当于只能达到最佳配置0110的加速比1.37的78%。

5．总结

本论文展示了一种帮助使用者将系统硬件预取器的效率提升至最大的框架。我们在试验中使用的处理器带有4个预取器，因此共有16种可能的预取配置。我们的框架对这16种组合进行了详细的研究，并且记录了每种配置在训练程序集中的性能。框架使用硬件性能事件来描述代码，以达到找出新输入程序和训练程序之间相似性的目的。由于现代处理器已经支持数百种性能事件，我们的框架使用了剪枝算法来构造简明的特征集。特征集中组合了多种预取器配置的性能，被用来训练三种机器学习模型。其中基于欧几里得距离的模型是我们为推出预取器配置设计的特殊模型。此模型能够在PARSEC标准程序组和两个额外程序上，推出平均96%的可达的预取加速比。

6. 参考文献

1. [] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications,” in Proceedings of the 17th international conference on Parallel architectures and compilation techniques. ACM, 2008, pp. 72–81. [↑](#endnote-ref-1)
2. [] R. L. Lee, P.-C. Yew, and D. H. Lawrie, “Multiprocessor cache design considerations,” in Proceedings of the 14th annual international symposium on Computer architecture. ACM, 1987, pp. 253–262. [↑](#endnote-ref-2)
3. [] T. Chen and J. Baer, “A performance study of software and hardware data prefetching schemes,” in Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on. IEEE, 1994, pp.223–232. [↑](#endnote-ref-3)
4. [] H. Kang and J. L. Wong, “To hardware prefetch or not to prefetch?:a virtualized environment study and core binding approach,” in ACM SIGPLAN Notices, vol. 48, no. 4. ACM, 2013, pp. 357–368. [↑](#endnote-ref-4)
5. [] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, “Prefetch-aware shared resource management for multi-core systems,” ACM SIGARCH Computer Architecture News, vol. 39, no. 3, pp. 141–152, 2011. [↑](#endnote-ref-5)
6. [] T. R. Puzak, A. Hartstein, P. G. Emma, and V. Srinivasan, “When prefetching improves/degrades performance,” in Proceedings of the 2nd conference on Computing frontiers. ACM, 2005, pp. 342–352. [↑](#endnote-ref-6)
7. [] J. Lee, H. Kim, and R. Vuduc, “When prefetching works, when it doesn’t, and why,” ACM Transactions on Architecture and Code Optimization (TACO), vol. 9, no. 1, p. 2, 2012. [↑](#endnote-ref-7)
8. [] S. Jayasena, S. Amarasinghe, A. Abeyweera, G. Amarasinghe,H. De Silva, S. Rathnayake, X. Meng, and Y. Liu, “Detection of false sharing using machine learning,” in Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and

   Analysis. ACM, 2013, p. 30. [↑](#endnote-ref-8)
9. [] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O’Boyle, and O. Temam, “Rapidly selecting good compiler optimizations using performance counters,” in Code Generation and Optimization, 2007. CGO’07. International Symposium on. IEEE, 2007, pp. 185–197. [↑](#endnote-ref-9)
10. [] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks,B. Mendelson, E. Bonilla, J. Thomson, H. Leather et al., “Milepost gcc:machine learning based research compiler,” in GCC Summit, 2008. [↑](#endnote-ref-10)
11. [] J. Demme and S. Sethumadhavan, “Approximate graph clustering for program characterization,” ACM Transactions on Architecture and Code Optimization (TACO), vol. 8, no. 4, p. 21, 2012. [↑](#endnote-ref-11)
12. [] C. McCurdy, G. Marin, and J. Vetter, “Characterizing the impact of prefetching on scientific application performance,” in International Workshop on Performance Modeling, Benchmarking and Simulation of HPC Systems (PMBS13), 2013. [↑](#endnote-ref-12)
13. [] S. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, “Machine learning-based prefetch optimization for data center applications,”in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. ACM, 2009, p. 56. [↑](#endnote-ref-13)
14. [] J. Treibig, G. Hager, and G. Wellein, “Likwid: A lightweight performance-oriented tool suite for x86 multicore environments,” in Parallel Processing Workshops (ICPPW), 2010 39th International Conference on. IEEE, 2010, pp. 207–216. [↑](#endnote-ref-14)