## Domino Temporal Data Prefetcher

### 出处与作者

出处：HPCA ’18

作者：**Mohammad Bakhshalipour**, Department of Computer Engineering, Sharif

University of Technology ;

**Pejman Lotfi-Kamran**, School of Computer Science, Institute for Research

in Fundamental Sciences (IPM) ;

**Hamid Sarbazi-Azad**, Department of Computer Engineering, Sharif University of Technology ; School of Computer Science, Institute for Research

in Fundamental Sciences (IPM)

### 背景与问题

缓存未命中现象会因为等待数据的到来而导致空闲等待，这使得处理器无法完全发挥其最佳性能。实验观察到，这种实际性能与其潜能空间之间的差距是由预取器的查找机制造成的。目前最新的时间预取器有通过单个未命中地址确定预取流的STMS和通过两个未命中地址确定预取流的Digram。研究发现，STMS通过单个未命中地址确定预取流时会出现混淆两个以相同未命中地址开头的预取流。而若通过两个未命中地址确定预取流，虽其预取准确性大大提高，但未命中地址的预取覆盖率却有所降低，综合来看，与STMS相比，Digram的性能并未获得多少提升。另外，通过上述预取器实现有效预取需在每个未命中发生时进行两次主存请求，带来了很大的延迟。

### 方案设计

本文提出了一种新型时间预取器Domino。将缓存未命中及预取命中视作为触发事件。在每次发生未命中时，Domino预取器根据与其相关的两次触发事件或者本次触发事件进行历史查找以找到匹配的预取流。若通过与其相关的两次触发事件成功匹配，Domino则使用该匹配的后续流进行预取；若通过本次触发事件成功匹配，则仅预取匹配处后面的第一个地址。若触发事件是预取命中，Domino继续预取（仍是同一个预取流，该预取流被称作活动流）；若触发事件是缓存未命中，则另寻预取流。

Domino依赖EIT表和HT表实现上述机制，它针对每个核在物理地址空间中开辟有连续空间以存储EIT表和HT表。其中，HT表存储有该核中观察到的未命中的序列，EIT表存储有一个t (a,p)结构对，该结构对表示未命中地址t的后一个未命中地址是a，指向该未命中短序列(t,a)在HT表中最后一次的出现的指针是p。我们将未命中地址t及其相关“地址-指针”对称为super-entry，并将“地址-指针”对称为entry。EIT表和HT表的结构及其细节如图1.1所示，Domino中EIT表和HT表与STMS及Digram中EIT表和HT表的对比如图1.2所示。

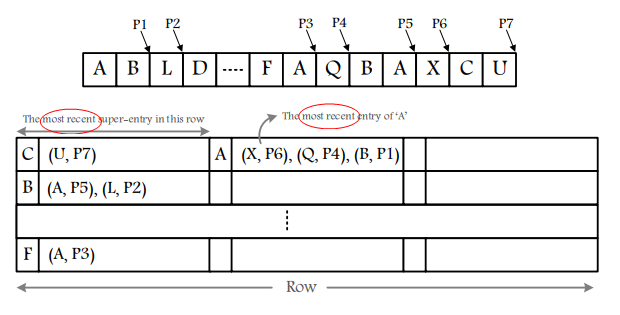


图1.1 Domino中EIT表和HT表结构

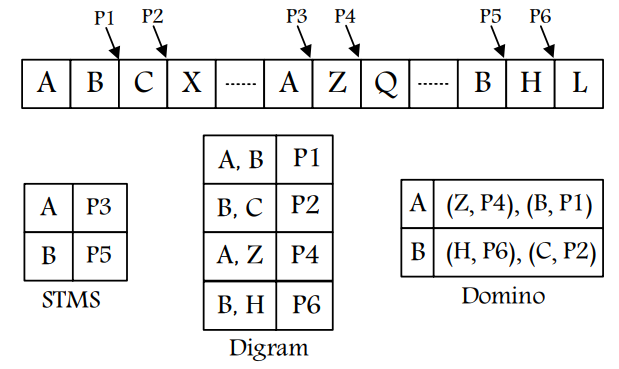


图1.2 Domino中EIT表和HT表与STMS及Digram中EIT表和HT表的对比

Domino受益于以下四个存储缓冲区：

①LogMiss：记录触发事件的序列；

②FetchBuf：更新时存储EIT表的某一行；

③Perfetch Buffer：存有预取缓冲块；

④PointBuf：存有活动流的地址序列。

Domino采用统计更新。对于每一次触发事件，将其地址存入LogMiss中，LogMiss大小为两个缓冲块，当其中一个存满后，统一将一系列未命中地址写入HT表中。对于EIT表的更新，对上述N个触发事件中的任一个，将EIT表中与其相关的那一行写入FetchBuf中。若该行中无匹配super-entry，则写入新的super-entry。若该行中有匹配super-entry，但无匹配entry，则在该super-entry下写入新的entry，并更新新指针。该行完成更新后再统一写入主存中的EIT表中。

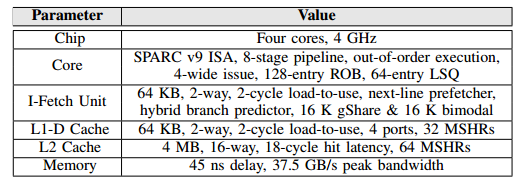
对于查找及预取机制，若触发事件是预取命中，则利用PointBuf继续预取Perfetch Buffer中的活动流；若触发事件是缓存未命中，则需查找新的预取流。对于新预取流的查找，Domino将对应于该未命中地址的EIT表的那一行写入PointBuf中并查找匹配的super-entry，若无匹配，则略过；若有匹配，则将该super-entry下的最新entry写入Perfetch Buffer中进行预取。当下一个触发事件发生时，若该触发事件是预取命中，则利用PointBuf继续预取Perfetch Buffer中的活动流；若该触发事件是缓存未命中，则在该super-entry下查找匹配entry（可能不再是上述的最新entry），若无匹配则丢弃该预取流并利用此未命中地址将其对应的EIT表的那一行写入PointBuf中，若有匹配，则据此创建活动流并将其从HT表中写入PointBuf中进行预取。

需要注意的是，Domino优先进行查找及预取机制，仅在查找及预取机制完成时进行必要的更新。

### 实验及结果

**实验方法与设置：**实验设备：Flexus全系统时间仿真器。实验参数如表1.1所示。

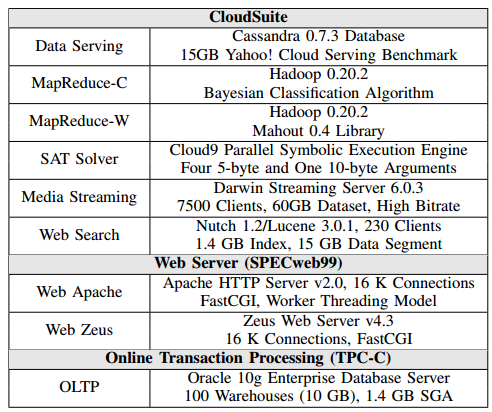
表1.1 试验参数



工作负载参数如表1.2所示。

实验所用基线指无数据预取，做对照的预取器有基于空间预取的VLDP、基于PC局域性的ISB、基于全局未命中及单个地址查找的STMS、基于两个未命中地址进行查找的Digram、本文所提出的Domino和用于衡量预取潜能空间的Sequitur。

表1.2 工作负载参数



**实验结果：**图1.3显示了各预取器在预取等级为1的情况下的覆盖率及过预测情况。其中，已覆盖未命中指的是通过预取成功消除的未命中，过预测指的是错误的预取，这可能会造成带宽开销。可以观察到，Domino的覆盖率仅低于理想Sequitur的覆盖率且其过预测情况也较为良好。

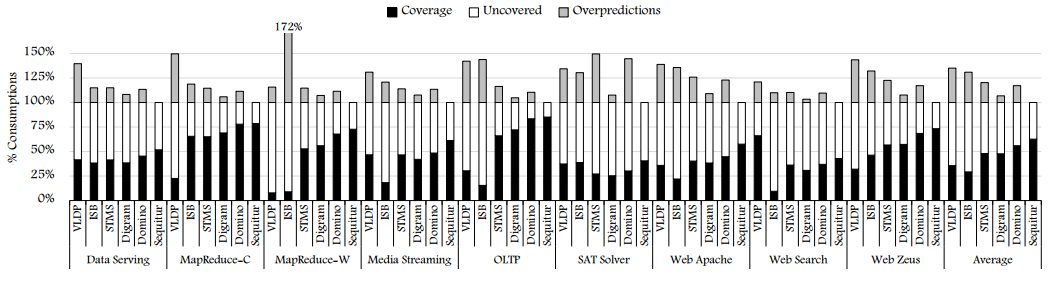


图1.3 各预取器在预取等级为1的情况下的覆盖率及过预测情况

图1.4显示了各预取器在预取等级为4的情况下的覆盖率及过预测情况。可以观察到，Domino的覆盖率高于其他预取器的覆盖率且其过预测情况明显优于其他预取器。

图1.5显示了Domino相较于基线的性能提升，并与ISB、VLDP、STMS及Digram进行比较。可以观察到，Domino性能得到了明显的提高。

图1.6显示了Domino相较于基线的开销，并与STMS及Digram进行比较。可以观察到，Domino开销较低。

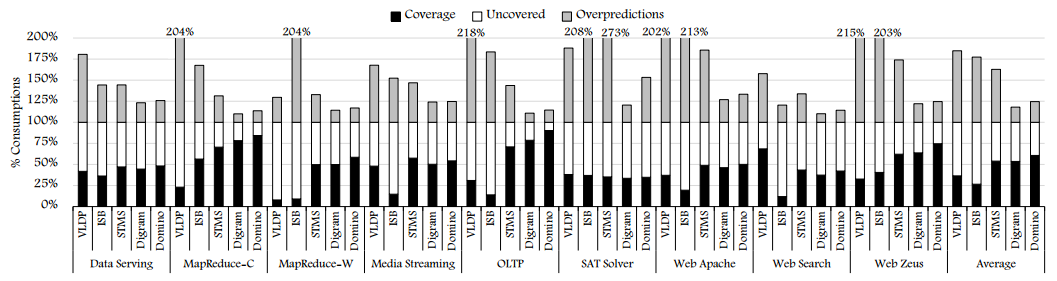


图1.4 各预取器在预取等级为4的情况下的覆盖率及过预测情况

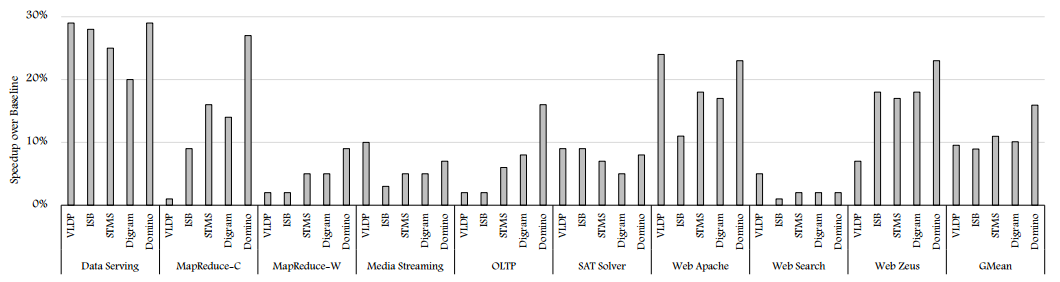


图1.5 各预取器相较于基线的性能提升

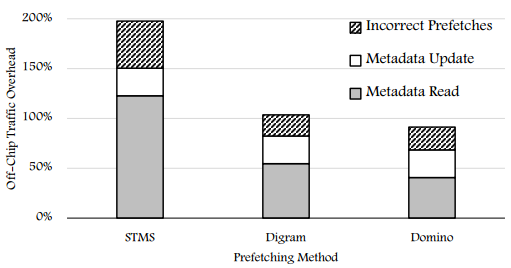


图1.6 各预取器相较于基线的开销

## 2）Bingo Spatial Data Prefetcher

### 出处与作者

出处：HPCA ’19

作者：**Mohammad Bakhshalipour**, Department of Computer Engineering, Sharif

University of Technology ; School of Computer Science, Institute for Research

in Fundamental Sciences (IPM) ;

**Mehran Shakerinava**, Department of Computer Engineering, Sharif University of Technology ;

**Pejman Lotfi-Kamran**, School of Computer Science, Institute for Research

in Fundamental Sciences (IPM) ;

**Hamid Sarbazi-Azad**, Department of Computer Engineering, Sharif University of Technology ; School of Computer Science, Institute for Research

in Fundamental Sciences (IPM)

### 背景与问题

片外存储器访问的高延迟使得处理器无法完全发挥其最佳性能。通过预测未来的存储访问并在处理器显式发布该请求前预取不位于缓存中的存储访问，数据预取机制被用来弥补处理器与存储器之间的性能差距，虽有成效但仍有上升空间。空间预取器依赖空间地址相关性实现对未来存储访问的预测。当应用请求某一页面时，空间预取器观察该页面的全部访问，据此记录一个footprint，同时将此footprint分配给相应的事件（event），并将上述两个记录以<event,footprint>形式存储于历史表中以供预取器进行预测。实验发现，基于短事件（较少事件的组合）的预取具有较高的匹配可能性，但准确性较低；基于长事件（较多事件的组合）的预取具有较高的准确性，但匹配可能性较低。单纯依赖于短事件或长事件进行预取是最新空间预取器性能不佳的主要原因。研究发现，TAGE型预测器（查找顺序是：最长事件→第二长事件→···→最短事件）既利用短事件又利用长事件进行预测，兼顾匹配可能性及准确性。但其使用多个历史表进行记录，当发生新触发事件时，需要对每个历史表进行更新，大大增加了开销。

### 方案设计

本文提出了一种新型空间预取器Bingo，它根据实验结果，采用两种长度的事件：PC+Address及PC+Offset。Bingo利用“短事件是包含在长事件中的”这一特点，仅使用单个历史表，对于每个触发事件采用多次查找，兼顾不同长度的事件，兼顾匹配可能性及准确性。

Bingo预取器结构如图2.1所示，其历史表采用PC+Offset的哈希值索引以确定其位置，并采用PC+Address做标识以供查找。当触发访问发生时，采用该访问PC+Offset的哈希值确定其位置，由于哈希转换是一种压缩映射，此时确定的是一个集合；再根据该访问PC+Address进行第一次查找，若有匹配则预取；若无匹配则进一步根据PC+Offset在此集合内进行第二次查找，若有匹配则预取。

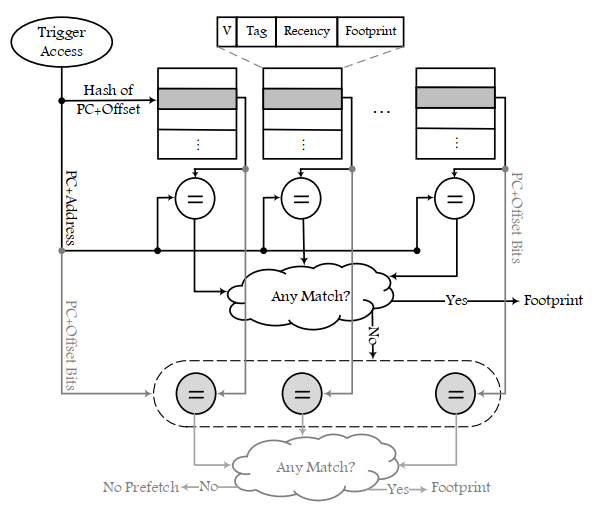


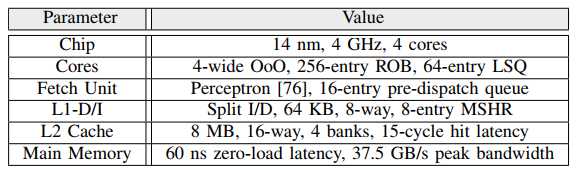
图2.1 Bingo预取器结构

由于第二次查找的事件相对较短，准确性有所下降，Bingo预取器可能会有多个匹配，此时可根据条目中的“Recency”项选取最近使用过的信息进行预取，也可选取对于所有匹配项覆盖率最广的信息进行预取，实验发现，后者性能更佳。

### 实验及结果

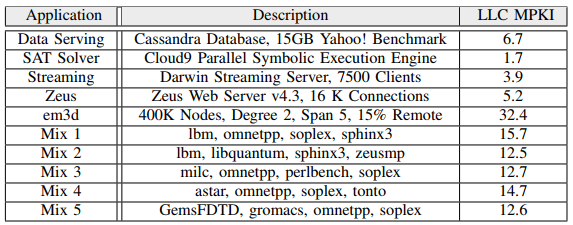
**实验方法与设置：**实验设备：ChampSim。实验参数如表2.1所示。

表2.1 试验参数



工作负载参数如表2.2所示。

表2.2 工作负载参数



实验所用基线指无数据预取，做对照的预取器通过单个偏移进行预测以提高预取及时性的BOP、基于增量偏移的SPP、基于多历史表的VLDP、利用存储访问映射标记最近访问的AMPM、本文方法的基础SMS以及本文所提出的Bingo。

**实验结果：**图2.2显示了各预取器的覆盖率及过预测情况。其中，已覆盖未命中指的是通过预取成功消除的未命中，过预测指的是错误的预取，这可能会造成带宽开销。可以观察到，Bingo提供有最高的覆盖率且其过预测情况也较为良好。

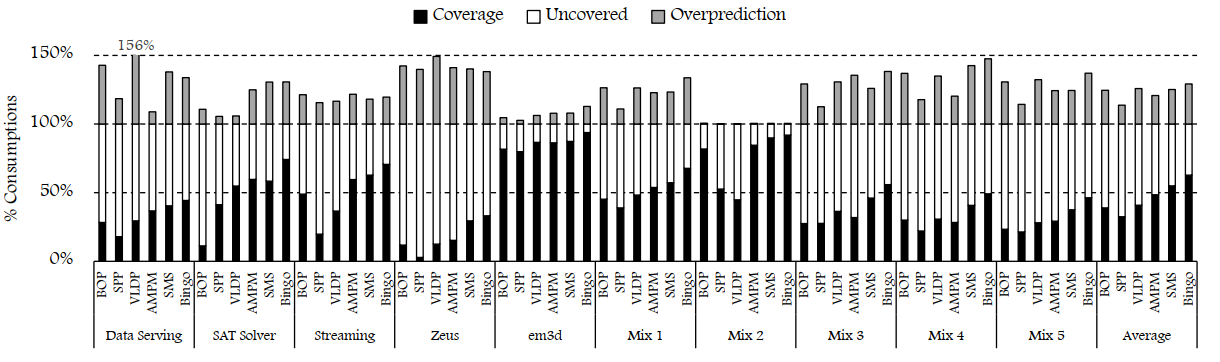


图2.2 各预取器在预取等级为1的情况下的覆盖率及过预测情况

图2.3显示了各预取器的性能提升情况（与基线比较）。可以观察到，Bingo性能提升最高。

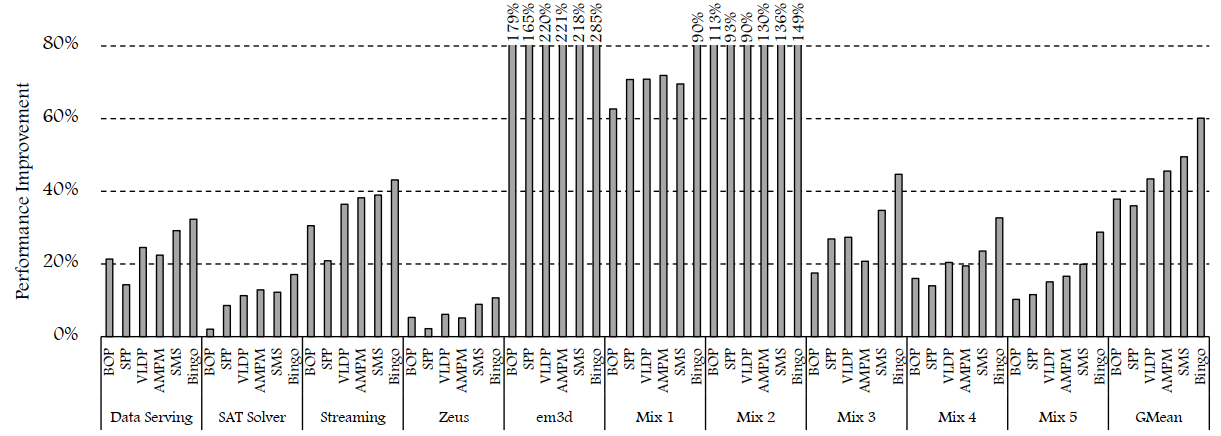


图2.3 各预取器的性能提升情况

图2.4显示了各预取器的性能密度（即单位面积的吞吐量）的提升情况，以此指标能观察该预取器是否真正做到了性能增益大于其面积开销。可以观察到，Bingo的性能密度提升最高。

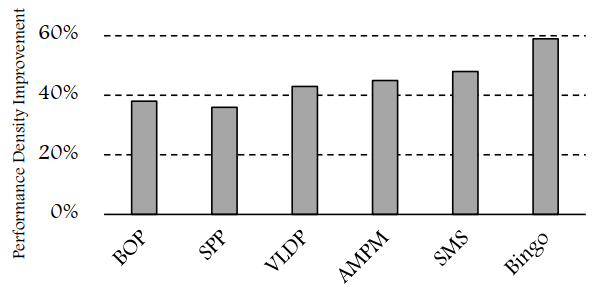


图2.4 各预取器的性能密度提升情况

## 3） Exploring System Challenges of Ultra-Low Latency Solid State Drives

### a) 出处与作者

出处：HotStorage ’18

作者：**Sungjoon Koh**, Computer Architecture and Memory Systems Laboratory,

Yonsei University;

**Changrim Lee**, Computer Architecture and Memory Systems Laboratory,

Yonsei University;

**Miryeong Kwon**, Computer Architecture and Memory Systems Laboratory,

Yonsei University;

**Myoungsoo Jung**, Computer Architecture and Memory Systems Laboratory, Yonsei University

### b) 背景与问题

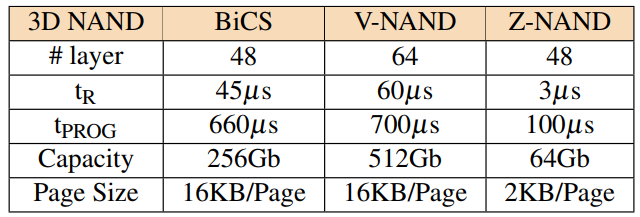
因诸如内部并行性、I/O排队/调度策略和DRAM缓冲区等不同的体系结构支持，现代SSD能够满足高带宽的要求。然而，为缩短I/O操作基本单元的延迟时间，需要进行低级别的内存设计更改和设备更新。Z-NAND能提供超低延迟（此类SSD统称为ULL SSD），它利用基于单级单元(SLC)的3D闪存设计，对诸如默认的I/O页面大小和DDR接口进行了I/O电路优化，以分别支持闪存访问和数据传输延迟的低延迟。业界文章揭示了ULL SSD的低级别设备性能，但由于这些新型SSD还未能在公共市场上使用，很难通过考虑不同的系统执行参数来预测它们的实际性能。本文描述了真实800GB Z-SSD原型的性能行为，并分析了将ULL SSD集成到当前软件存储堆栈中的系统级挑战。

### c) 方案设计

表3.1总结了3种最新3D闪存技术的设备级特性。Z-NAND使用48个堆叠字线层，分别提供3µs和100µs的读取延迟和写入延迟。虽然Z-NAND的存储容量和页面大小较小，但通过将更多的Z-NAND包放入设备以作为扩展解决方案，可以提供具有较短延迟的大容量存储卷。

图3.1是ULL SSD的内部结构及分离DMA结构。其中，分离DMA管理结构通过低粒度数据访问受益于更低的延迟。高端SSD存储结构包括被称为channel的多系统总线及其用于传递各个闪存包的多个数据路径，被称为way。表3.1中可知Z-NAND基本I/O单元大小较小，能够提供更细粒度的操作。ULL SSD将4KB请求拆分为两个2KB操作并同时送入两个不同的channel，这两个不同的channel被绑定为一组，称为super-channel。分离DMA负责此过程，且

表3.1 3种最新3D闪存技术的设备级特性



若super-channel的某一channel出现坏块，为避免存储空间的浪费，配有remap checker将坏块与干净块进行重映射。

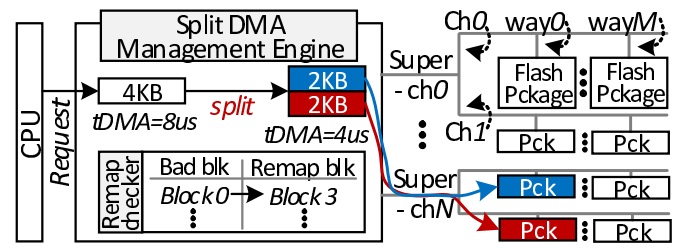


图3.1 ULL SSD的内部结构及分离DMA结构

另外，为防止资源冲突，即读取某页数据时，若该页对应super-channel正忙于更早下发的写操作，则读操作将等待，这将增加延迟，ULL SSD通过将写操作的精确点存储于控制器一侧的小缓存中，进而挂起写操作，使读操作先完成，避免上述延迟；待读操作完成后再进行写操作（读操作所需时间很短，对于写操作来说可忽略其影响），以进一步提供更低的读写延迟。

### d) 实验及结果

**实验方法与设置：**

**①基准：**使用FIO v2.99作为基准套件以描述NVMe和ULL SSD，且为所有评估设置了一个O\_DIRECT标志以绕过页面缓存，并直接向/从底层SSD提供I/O请求。另外还使用AIO(livaio)作为I/O引擎来生成异步块I/O请求。尽管测试了不同块I/O大小(从4KB到32KB)的SSD，但对于性能分析和CPU利用率等特定评估，默认的块大小配置为4KB。另一方面，通过I/O引擎启用同步preadv2/pwritev2(pvsync2)，以分析不同类型的I/O完成方法带来的系统影响。

**①设备配置：**4GHz、4芯英特尔i7处理器(i7-4790K)及16GB DDR4 DRAM试验台。测试中，所有SSD都通过PCIe 4x 3.0通道连接到试验台的主机上。在本研究中，使用了Ubuntu 16.04 LTS和Linux kernel 4.14.10，其中包含了最新版本的NVMe存储堆栈。最后，使用collectd(5.5.1)来分析CPU周期，使用英特尔Vtune放大器2018测量NVMe存储堆栈显示的内存边界。

**实验结果：**图3.2显示了不同I/O深度下ULL SSD和NVMe SSD的总体延迟特性。如图3.2(a)所示，随着队列深度的增加，NVMe SSD的执行时间特性明显恶化。相反，即使在具有大量I/O队列的测试中，ULL SSD也提供了相当可持续的性能。如图3.2(b)所示，NVMe SSD和ULL SSD之间的这种性能差异在长尾延迟特性中更为明显。

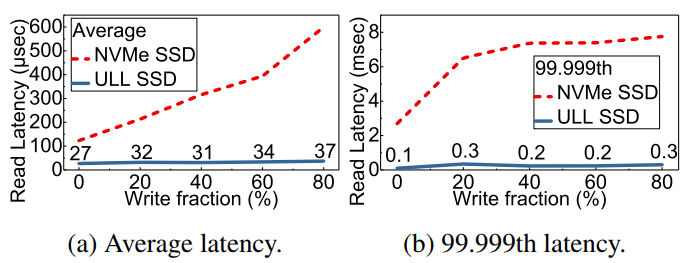


图3.2 不同I/O深度下ULL SSD和NVMe SSD的总体延迟特性

图3.3显示了不同I/O深度下ULL SSD和NVMe SSD的带宽特性。可以观察到，ULL SSD在I/O深度较低时即可达到最大带宽利用率。

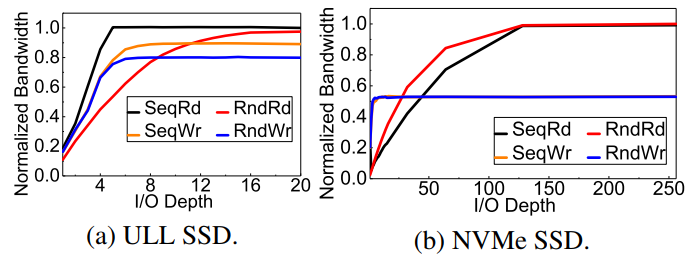


图3.3 不同I/O深度下ULL SSD和NVMe SSD的带宽特性

图3.4分析了读取和写入混合时的I/O干扰的程度。横坐标表示读写操作中写操作的占比，可以观察到，虽写操作占比的增大，NVMe SSD干扰增大，但ULL SSD抗干扰性能较好。

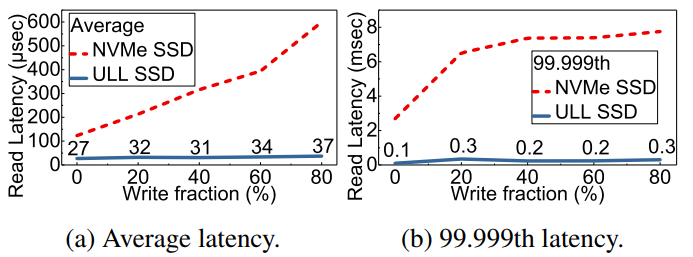


图3.4 读取和写入混合时的I/O干扰的程度

图3.5显示了基于轮询及基于中断的ULL SSD和NVMe SSD的延迟特性。可以观察到，NVMe SSD中两者并无太大差别，但ULL SSD中采用基于轮询的I/O延迟更低。然而，由于轮询(例如高CPU周期和频繁的存储器访问)传递的系统级开销招致许多CPU停滞，将消耗比基于中断的存储系统更大的系统功率。

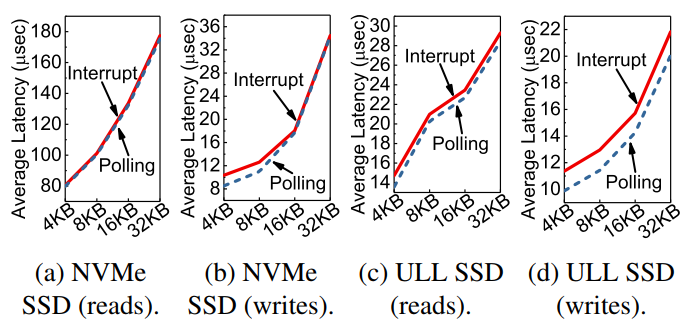


图3.5 基于轮询及基于中断的ULL SSD和NVMe SSD的延迟特性

## 4）DenseFS: A Cache-Compact Filesystem

### a) 出处与作者

出处：HotStorage ’18

作者：**Zev Weiss**, University of Wisconsin-Madison;

**Andrea C. Arpaci-Dusseau**, University of Wisconsin-Madison;

**Remzi H. Arpaci-Dusseau**, University of Wisconsin-Madison

### b) 背景与问题

通过收集执行各种元数据操作的现有文件系统（btrfs、ext4、f2fs、tmpfs和xfs）的指令级动态trace，文章发现：①指令高速缓存占用通常约是数据缓存占用的两倍。相对于当前x86处理器中的第一级高速缓存的大小，两者都足够大，以至于会显著干扰由应用建立的暖用户空间第一级高速缓存的状态；②许多数据缓存访问相对来说是浪费的，因为它们以整行为单位取代缓存中的另一行)，只提供少量字节（通常是单个内存访问）。这类访问既不显示缓存优化的空间局部性，也不显示其时间局部性，因此无法很好地利用它们；③由于默认情况下执行本身是连续的，指令访问在一定程度上减少了对缓存资源的浪费。然而，这种空间局部性相对较小。鉴于指令缓存占用较大，这仍然不是对硬件资源特别有效的使用。综上，希望能够实现一种更小缓存占用的文件系统。

### c) 方案设计

为实现更小的缓存占用，DenseFS从数据缓存压缩和指令缓存压缩两方面着手。数据缓存压缩方面：①因为应用程序很少实际使用访问时间(文件系统通常使用noatime选项挂载)，用Linux内核的内部8字节ktime\_t替换了庞大的16字节结构timspec，并完全删除了atime成员。通过将时间戳上的空间从48字节减少到16字节，节省了32字节;②inode编号的使用也相对较少，因为其编码的唯一信息是唯一标识符，则可在不影响功能或语义的情况下删除。 DenseFS的stat调用使用从inode本身的内存地址派生的值来填充st\_ino字段，而不是在每个inode中存储inode编号。为使这些合成的inode数保持持久减去DenseFS内存区域的基址，形成一个偏移（而非原始指针值），然后使用存储在DenseFS超级块中的一个密钥对此偏移量进行XOR，以避免敏感的元数据泄漏到用户空间的可能；③因为用户、组和模式字段包含非常小的熵，则在每个单独的索引节点中对该信息进行编码是对空间非常低效的使用。在DenseFS中，通过保持文件系统范围的元组并用一个16位索引替换索引节点结构中相应的三个条目来压缩此信息，节省了另外的10个字节。

指令缓存压缩方面：①函数对齐：函数的代码足够短且适合单个缓存行时，其起始地址有可能与缓存线边界间存在偏移，使其溢出到下一行中，则相较于真实缓存行需求，其将执行更多行。通过在64字节的边界上进行对齐能够避免这一陷阱，并将函数保存在单个高速缓存行中；②分支提示：条件判断代码中，若执行条件出现的可能性较低，则此代码块将很少被执行，导致空间的浪费。通过对此类分支的识别，可以将适当的注释添加条件判断代码中，并减少缓存空间宝贵字节的浪费；③函数排序：多个函数组合即使是在函数对齐后也会因代码分布布局而占用比实际所需空间更多的缓冲行，通过函数排序将它们组合在定义它们的源文件（lib / string.c）中，能够实现减少空间浪费的期望结果。

### d) 实验及结果

**实验方法与设置：**使用精细参数化的合成微基准评估DenseFS；所有测量均在4.13系列Linux内核的英特尔至强E5-2670 CPU上进行。

**实验结果：**图4.1显示了基准性能结果。纵坐标表示当在给定文件系统上插入对给定系统调用的常规调用时，执行用户模式代码所花费的时间的相对增量(即syscall在用户模式执行上的性能损失)，横坐标表示在系统调用之间执行的用户模式代码的数据和指令缓存占用。可以观察到，DenseFS性能优于其他现有文件系统。

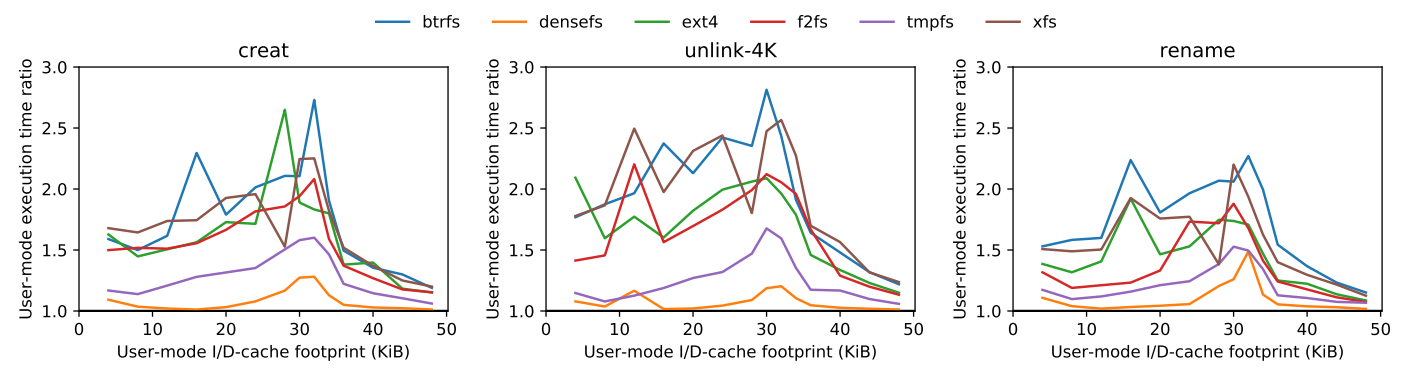


图4.1 基准性能结果