**一个丰富跨层抽象的实例：通过Expressive Memory缩小语义差距**

**关键词：语义gap、跨层抽象、存储优化实例（cache+DRAM）、XMem=Atom+Interface、软件模拟**

**①atom承载来自应用程序不同的语义，XMem向上有与Application、向下有与Hardware的接口；**

**②对于cache优化，根据atom携带数据的使用率装填cache，作cache的替换算法的优化；**

**③对于DRAM的优化，根据atom携带的不同的访问模式、数据结构属性提供给OS和访存控制器，两者根据语义选择更加聪明的bank、channel和row。**

**Abstract**

这篇文章介绍了一个跨层的接口——Expressive Memory(XMem)来实现应用程序和系统软件、硬件架构之间的语意通讯。XMem提供了一个易适应的、易扩展的抽象——叫做Atom，它能够使得应用程序表达关键的程序语意，比如程序如何访问数据以及数据本身的属性，同时能够将这些语意传递给操作系统和硬件架构。通过提供一些别的方式不能获得的关键信息，XMem为操作系统和硬件架构提供了一个程序数据全新的、丰富的视角，这有利于优化存储系统性能（比如cache、存储控制器等）。

通过给应用程序和存储资源缩小差距，XMem提供了两个好处。第一，它使得结构级/系统级的技术开始leverage（举债经营、借债收购）关键程序的语意，这些语意往往难以预测和推断的；第二，通过减轻缓和代码适应具体硬件的需求，XMem改善了软件优化的功效和移植性。虽然XMem当初是设计用来优化主存的，我们还证明了XMem的好处可以应用于以下两个情境：（1）基于软件的cache移植性优化，在优化过程中通过传达数据位置实现；（2）主存中基于操作系统的页放置通过lervage数据结构的语义和它们的访问语义。

**1.Introduction**

传统来说，软件栈和体系结构（ISA和虚拟存储）主要是设计用来传达程序功能以确保程序能够按照软件的需求来运行。一个应用程序被转化成ISA指令和一系列对虚拟存储的访问操作，同时，由于只剥落成了保证能够正确执行的基本语义，程序本身的更高层语义会被丢失。比如：应用程序的数据结构是一种高层的语义，在编译生成的ISA指令交给操作系统or硬件后就会丧失这些信息，然而数据结构这种高层语义对于底层的性能优化来说非常有用。

所以，换句话说，我们为了提升性能会把应用程序的高层语义传递给系统级or体系结构级，两者之间存在的语义gap使得语义丢失。在方方面面缩小关注点，本论文只把注意力放在存储子系统上。现代计算机系统在存储性能方面做了很多优化，比如：预取（DRAM技术中的Fast Page、Burst等）、cache技术、存储控制器等。语义gap主要体现在以下两个方面：

含义1.OS和存储子系统被迫去预测和推断程序行为。存在的挑战如下：（i）每个部件(例如L1 Cache,存储控制器)只能有数据访问的局部视野，而看不到全部（ii）专用硬件要求对所有的部件都进行优化（iii）优化是有用的只有当程序行为不是先验的。

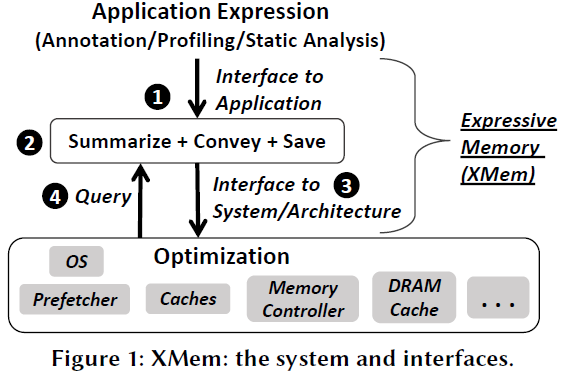
含义2.由于存储资源的可用性在变化（虚拟存储、多线程），软件优化中作准确的假定会很困难，这对性能的portability(便携性)带来挑战。

有很多历史上提出的细粒度层上的软硬件协同的技术——新的ISA指令系统[1-13]、程序注释以传达程序语义[3、14-17]、针对具体优化的软硬件协同设计[18-22]。这些方法有两个明显的不足：1.这些优化方法只针对特殊的具体的问题，不具有一般性；2.使用特殊的指令指导特定的部件进行特定的行为，对不同微体系结构不具有移植性和可编程性。

**Our Goal.**我们首先问自己这样一个问题：能否设计一个统一的抽象层次和接口来在应用程序和系统级部件之间传达关键的程序语义。对此，我们提出了XMem(Expressive Memory)这样一个统一的、低开销的、可扩展的跨层接口，并解决了诸多挑战（这些挑战是普适性、开销、性能、可编程性等多方面的tradeoff）。

**Expressive Memory**包含以下两个部件。

1. The Atom.是我们提出的一个新的软硬件的抽象，每个Atom映射到具有相同语义属性的数据上，程序显示地指明与OS和Hardware打交道的Atoms。Atom携带数据属性(类型、稀疏性)、访问属性(读或写)、数据局部性这3种程序信息，并在程序执行期间追踪数据属性。
2. System and Cross-layer Interfaces(接口).Figure 1展示了它的组成：（i）圆圈①应用程序的接口使得软件能够将程序注释、静态编译分析、动态剖视等信息显示地传达Atom到XMem系统；（ii）圆圈②表示XMem系统能总结、传达、保存Atom；（iii）圆圈③表示该接口提供关键的补充信息给OS和结构级部件；圆圈④表示接口支持系统/结构级部件反向查询XMem获得更高层的语义。

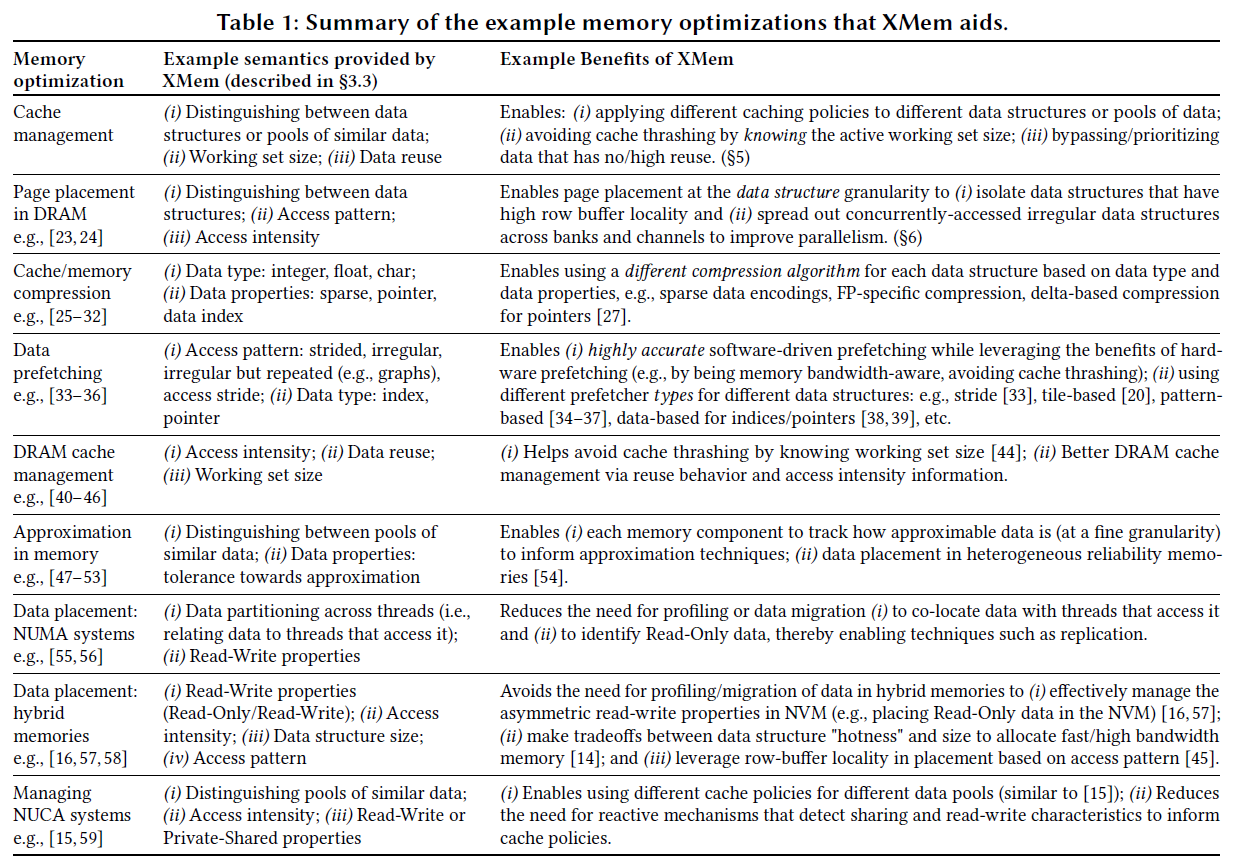


**User Case.**Xmem设计成一个统一的、提供广泛存储优化的接口。本篇文章，我们首先证明了XMem系统对基于软件的cache优化方法带来的益处。通过增加数据局部性的语义（Atom信息可以携带并传递给cache）,cache的管理和预取策略就会提升，从而增加了cache的命中率和优化portability，性能损失从55%下降到了6%；第二，我们证明了XMem系统通过传达数据结构和访问语义对于DRAM页管理带来的性能增益。XMem系统通过把规整的数据结构孤立、非规整的数据结构广播给DRAM的bank和channel从而最大化并行性，最终实验评估结果显示有平均8.5%的性能提升。

更一般地讲，Table 1展示了9种存储优化的方法以及XMem能够带来的好处。XMem的有点来源于3个方面。第一，通过使用同样的语义为各种各样的优化方法提供了统一、中央接口；第二，它把数据划分成多种语义类似的数据池，方便针对不同的数据池使用不同的策略；第三，通过提供更高层的语义提升了优化效果，这些高层的语义有的是每个部件能够获得的局部信息之外的全局信息，有的是难以精确推断或者需要剖视/检测才能去决定数据是只可读/只可写、私有/共享的。

本文主要做了一下方面的contribution：

* 本文是第一个尝试设计一个全面的、统一的跨层接口用以在高层程序和不同系统/体系结构部件之间进行语义交互的工作，并用于在现代微处理器中进行存储性能优化。
* 本文提出了由软硬件抽象Atom和跨层结构组成的XMem系统。它在覆盖各种程序语义和使用样例上更具有一般性和适用性，而且在不影响功能正确性的前提下提升性能，并对执行阶段数据属性的变化能做出反应，开销也小。
* 本文量化研究了使用XMem系统对于提升基于软件的cache优化的portability、基于OS的DRAM页管理效率，前者增加了数据局部性的语义，后者增加了数据结构和访问属性的语义，其他7中优化见Table 1。



1. **Goals and Challenges**

**2.1 Key Requirements**

此次设计考虑了以下几个必须条件：

(i)在原有功能不变的基础上作补充，新的接口不能影响源程序的正确性；

(ii)结构不可知。跨层抽象必须只基于源应用程序的特征而不是硬件/系统的特征，编写程序或软件时不需要精确知道cache大小、DRAM bank是否可获得等特性，都能进行性能优化；

(iii)通用性和可扩展性。接口需要能够传达各种程序语义给系统/硬件作性能优化；

(iv)低开销。接口必须在可控范围内维持低存储区域和低开销。

**2.2 Challenges**

目前系统和结构部件只能通过虚拟/物理地址看到源程序的数据，为了提供更高层的语义我们需要将每一个地址附加更多的语义信息。

**挑战1.**传达信息的粒度。最好的语义粒度是依赖于程序的，比如一个标题对应一个粒度，整个数据结构对应一个粒度。把所有的程序语义都映射到一个固定大小粒度的虚拟页会带来很大开销。

**挑战2.**通用性vs专用性。体系结构不可知要求应用程序或者程序员在传达更高层语义时对底层硬件结构和存储资源是不知晓的，造成传达的语义要不管理开销大，要么硬件部件难以使用。为了满足通用性，必须传达各种信息到所有不可知的部件，但为了满足专用性，具体部件又需要获取到简单可用的语义才能使用。

**挑战3.**修改数据语义。当程序执行时，数据结构和访问方式的语义会被修改，所有我们需要在程序运行期间传达动态的数据属性，并要克服由于运行时刻持续更新数据属性造成的数据开销。

**3.Our Approach:Expressive Memory**

Expressive Memory(XMem)是一个丰富的传达程序寓意的跨层接口，由2部分组成：一个有定义好的集合属性的软硬件抽象模型，一个给不同系统/存储传达和存储信息的接口集合。接下来我们将会描述XMem系统是如何解决以上3中挑战的。

**3.1 Atom Abstraction**

我们定义进行程序语义传递的基本单元——Atom，作为软硬件抽象的模型。Atom既作为信息表达的抽象，又作为传递、存储、检索信息的操作单元。应用程序在程序代码中能够动态的生成atom，每一个atom都描述了程序某个执行时刻具体的程序数据，然后OS和硬件结构将atom翻译成具体语义。

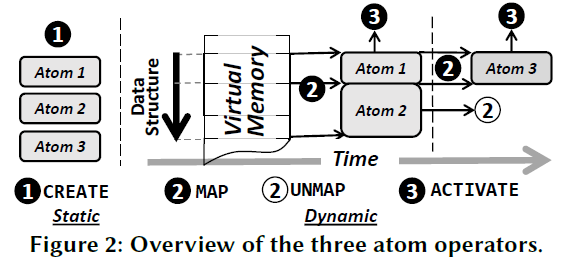
每个atom有三个关键组成部分：(i)属性：高层的数据语义；(ii)映射：虚拟存储空间(iii)状态：当前atom是活跃还是非活跃的状态。

**3.2 Semantic and invariant of an Atom**

* 均匀性：具有相同属性的数据要映射到同一个atom上。
* 虚拟地址到Atom多到一的映射：给定一个时间，任意一个虚拟地址(VA)只能映射到至多1个atom，系统/硬件便可以通过VA索引到唯一一个atom（如果有的话）。
* 不可改变的属性：虽然atom可以被动态创建，但是atom的属性在创建时便不可修改。在编译阶段每个atom描述的静态程序的语义可以提前传递给OS/硬件架构，而程序动态执行时，同一个数据对应的属性发生变化，只能创建新的atom，用新的atom的新属性描述数据的变化。
* 灵活的数据映射：一个Atom可以被映射到任意大小的数据集合（细粒度的），这很好的解决了挑战1.
* 激活/失活：虽然atom的属性是一成不变和静态创建的，但是atom本身在程序中可以动态的激活或失活。只有当atom是激活的，对应的属性才能被系统识别。程序执行过程中，当一个atom不在精确描述数据属性时，它就会被失活，同时一个新的atom被创建并映射到该数据。这使得程序在执行过程中，系统总是能通过atom看到正确的程序属性，即使通过程序静态创建的atom属性可以在编译/Load阶段提前传给系统。

为了管理Atom的3个组成部分，给出3个对应的操作通过二进制调用。(1)创建CREATE：支出静态确定的属性；（2）映射/解除映射：atom到对应的数据；（3）激活/失活：动态地告诉系统当前atom的属性是否对当前数据有效/失效。

Figure 2详细描述了以上3种操作。操作①表示根据程序代码静态创建Atoms；操作②表示当分配存储空间时，对应的atom会与一定范围的虚拟地址空间建立映射关系，操作③表示当建立银蛇的同时，所有的系统部件认为该atom的属性是有效的。反之，当程序执行阶段atom的属性不再精确传达对应数据的语义时，就会解除两者的映射关系(UNMAP)，并创建新的atom与对应数据建立映射关系。所以，映射/解除映射操作能够选择某个任意粒度的数据范围与对应atom建立或解除映射关系；激活/失活操作动态的有效/失效atom的属性。



**3.3 Attributes of Atom and Use Cases**

Atom属性包含以下3个：

1. 数据值属性：每一个Atom包含原数据的诸多属性，包括：数据类型(INT32、FLOAT32、CHAR8)和数据特性（稀疏、粗略、指针、索引）。
2. 访问属性：①访问模式（规整or不规整）②读写特性（只读、只写或可读可写）③访问密度：较其他数据的访问频率如何，用8个比特位表示，0表示最低的访问密度。
3. 数据局部性：工作集的大小和重复利用率（8个比特位表示，0表示没有被重复使用）是atom数据局部性中两个关键的属性。

需要注意的是，atom的抽象和接口传达的语义是灵活的、向前兼容的。以上Atom的属性是我们为了存储优化的目标挑选的，用来传递给OS/结构级部件。

**3.4 The XMem System:Key Design Choice**

接下来解释一下几个关键设计选择的合理性。

* 改变硬件支持。有两个原因：第一，我们的设计目标是最小化运行时刻细粒度地跟踪和检索语义带来的开销，所有我们改变硬件支持使得硬件执行几个关键的功能行为——映射/解除映射、激活/失活，这样做避免了由于频繁系统调用带来的高开销；第二，促使硬件管理的部件为XMem作改变，使得硬件支持能够更有效的传递关键语义到不同的硬件部分。
* 软件概述和硬件跟踪。这样可以最小化运行时刻复杂的软件开销。换句话说，静态的操作Create在执行之前的软件中处理（比如编译器），动态的映射/激活操作在运行时刻通过硬件处理。
* 中心化地全局跟踪和管理。所有的静态定义的atom都会被赋值一个全局的ID方便这个系统识别。
* 私有属性和属性翻译。应用程序传达的atom属性交给硬件部件（比如cache和prefetcher）作翻译是比较复杂的，为了解决这个挑战(挑战2)，当程序加载进来or上下文切换时，OS会触发一个硬件翻译的操作，把高层属性转换成具体的原始属性给每个硬件部件。

**3.5 XMem:Interface and Mechanism**

Figure 3详细描述了XMem整个系统部件的执行过程。

3.5.1 程序员/应用程序接口

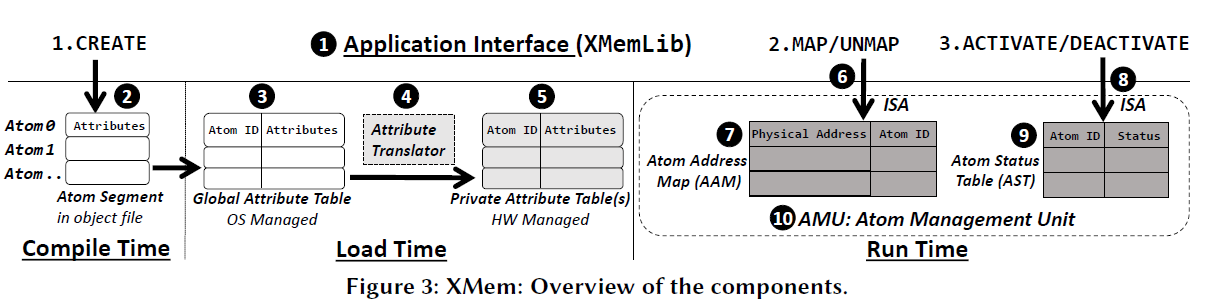
应用程序与XMemLib接口是通过库XMemLib。每个atom的数据结构中包含atom属性和操作函数（创建、映射、激活等），它的静态属性由编译器、程序员进行实例化赋值。

3.5.2 系统/结构级接口

XMemLib通过以下两种方式与OS/系统结构进行交互。

1. 在编译时编译器把静态的程序属性和atom关联起来并生成一张表，通过Atom ID去索引。在运行时刻，同一个静态atom会对应很多实例，所有创建同一个atom的调用会被映射到同一个静态的atom ID，这是可能的因为atom的属性在创建后是一成不变的。每一个atom映射的地址空间是不知道，因为虚拟地址空间在运行时刻才能被解析。圈②表明在编译时刻根据程序创建atom和atom属性的映射关系表并保存在目标文件的atom segment中。圈③是一个GAT（全局属性表,gloabal attribute table），当程序加载到内存被OS执行时，OS会读atom的segment，把每一个atom及其属性保存在GAT中，GAT在内核空间被OS管理。接下来OS会触发硬件翻译，将GAT中的高层属性翻译成每个硬件部件对应的硬件原语，并存储在PAT(私有属性表，private attribute table)中，交由每个硬件部件管理。
2. 在运行时刻，以XMemLib中函数调用的方式，XMem操作被翻译成ISA指令告知系统/硬件atom的行为，比如映射、激活/失活等。从概念上讲，圆圈⑥映射/解除映射被翻译成ISA指令去更新AAM（⑦ATOM地址映射,atom address mao），给每个atom ID关联一个物理地址PA，这里我们采用PA去索引AAM而不是VA是为了简化表的设计。圆圈⑧表示激活/失活操作，被翻译成ISA指令去更新AST（Atom状态表，Atom status table）中atom的状态。AST和ASM的管理交由AMU（Atom管理单元,Atom Management Unit）。

**灵活性和可扩展性**。系统/体系结构接口确保ISA和微体系结构只需要执行以上3中操作，而不需要口述哪些应用程序属性需要被传达。属性以二进制的形式存储在独立的原数据段中，并使用版本号去识别信息格式。信息格式会随着更多体系结构的增加而提升，版本号确保向前/向后兼容。未来新的体系结构在翻译语义时遇到老的XMem结构可以直接忽略其旧的信息格式，这样做保证了XMem的可扩展性。



1. **XMem：Detailed Design**

接下来我们详细说明XMem系统接口和组成的设计和执行细节。4.1节描述应用程序与OS、硬件结构之间的接口，4.2节描述XMem的关键部件，4.3节描述虚拟化环境中XMem的使用，4.4节描述设计的开销。

* 1. **The interfaces**
     1. 应用程序接口

XMem与应用程序的主要接口是二进制库XMemLib，它提供对atom的类型定义和功能调用。XMemLib包含对atom类的属性定义（见3.3节），对XMem的3种操作XMemLib以功能调用的形式提供，通过功能调用实现对atom属性、状态和映射关系的改变。Table 2总结了所有的功能调用。

（1）创建（CREATE）：函数CreateAtom根据输入参数创建atom、返回atom ID。一个程序中在同一时刻同时调用CreateAtom就会返回相同的atom ID.

（2）映射/解除映射(MAP/UNMAP)：输入atom ID和地址空间作为参数，产生对应的ISA指令告诉Atom管理单元（AMU）去更新Atom地址映射（Atom Address Map）。Atom2DMap支持二维的更新。

（3）激活/失活（ACTIVATE/DEACTIVE）：函数AtomActive和AtomDeactive作用于激活对应的atom，通过创建对应的ISA指令在运行时刻更新Atom Status Table。

* + 1. 操作系统接口

接口体现在两个方面。第一，操作系统管理GAT(全局属性表,global attribute table)，用于存储每个应用程序atom的属性。第二，OS能通过到存储分配器的接口选择性地查询虚拟地址范围和atom的静态映射。这个接口能确保OS在虚拟页映射到物理页之前知道这个映射，这样OS就能做类似于基于程序语义的存储放置等静态优化。具体来说，我们通过存储分配API（比如malloc）将Atom ID作为一个参数，存储管理器反过来将Atom ID作为一个参数经过系统调用传给OS。存储分配器通过返回虚拟页和它配套的Atom ID的方式维护atom和虚拟页的静态映射关系。编译器把A=malloc(size);和 AtomMap (atomID,A,size)装换成相应的API: A=malloc

(size,atomID)。AtomMap(atomID,A,size);这个接口使得OS能操纵虚拟到物理地址的映射，而不会造成其他的系统调用开销。

* + 1. 微体系结构接口

我们添加了两条ISA指令使得XMem系统能够在运行时刻跟硬件打交道。（I）ATOM\_MAP/ATOM\_UNMAP告诉AMU更新atom的地址（AMU用于管理atom地址映射和atom的状态），更新AAM。（II）ATOM\_ACTI-

-VATE/ATOM\_DEACTIVATE告诉AMU更新atom状态表（AST）。

* 1. **System Design:Key Components**

系统/接口通过三步获得跟访存地址相关的数据语义。（I）确定访存地址映射哪一个atom；（II）确定该atom是不是active的；（III）确定atom的属性；通过以下4个关键部件组成。

（1）atom地址映射(AAM，atom address map)。只保存最近的一次atom和它的PA。

（2) atom状态表（AST，atom status table）.通过Atom ID索引到对应状态。

（3）atom全局属性表和私有属性翻译器(GAT和PAT，attributes table和attributes translator)。GAT由OS存储，PAT由硬件部件存储，通过翻译器进行属性版本翻译。

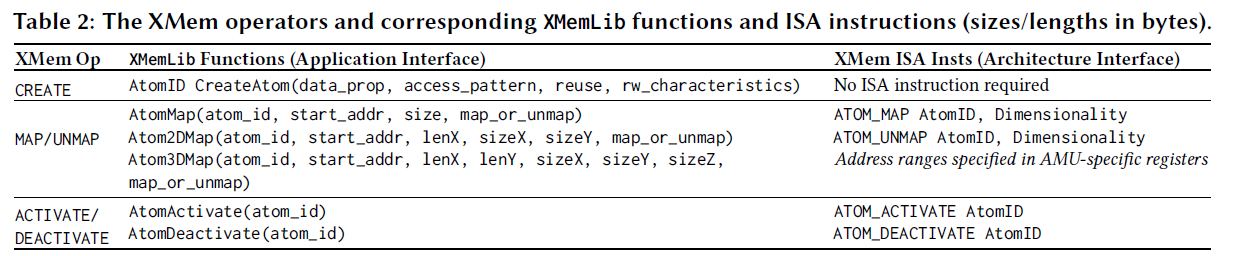
（4）atom管理单元（AMU，atom management unit）。负责AAM和AST的管理以及Atom ID 查询到对应的PA。AMU的设计思想和MMU很类似，如ALB(atom lookaside buffer)缓存Atom\_Lookup的查询请求。

**4.3 XMem in Virtualized Environments**

虚拟环境需要使用虚拟机或者在OS上层的容器。XMem能够无缝用在虚拟环境里。

XMem部件——AAM、AST、PAT、GAT这些关键部件无需做任何功能修改就可以在虚拟环境中运行。

XMem接口——创建、映射/解除映射、激活



/失活等接口也无需作任何操作修改就能在虚拟环境中运行。

优化方法。基于OS的软件优化需要OS看到可用的物理资源，硬件的优化需要硬件部件通过检索AAM和PAT获得atom的属性信息。

* 1. **Overhead Analysis(开销)**

整个XMem系统的开销包括以下4方面：存储开销、指令开销、硬件面积开销、上下文切换开销，这四种开销都是比较小的，详细的技术报考见参考文献[70]。

1. 存储开销包括atom的属性、状态和映射关系（AAM、AST、GAT、PAT）,4.2节已经说明每个表的大小都是足够小。
2. 指令开销指的是应用程序执行创建、映射/解除映射、激活/失活操作时会调用XMem指令。通过量化分析得出，额外增加的XMem指令开销最多不超过0.2%.
3. 硬件面积开销。我们增加了额外的两个硬件部件——属性翻译和AMU。使用CACTI 14nm工艺做评估发现增加的硬件面积是0.144mm2.
4. 上下文切换开销。XMem系统通过使用一个寄存器用来存储AST和GAT的指针来实现上下文切换，一个寄存器的开销是很小的，在上下文切换时会冲刷ALB和PAT，由于这些结构比较小所以开销也很小。

**5.Use Case 1：Cache Management**

**5.1 Overview**

XMem通过提供程序运行时刻难以推断的关键程序语义来补充软硬件之间的交互，本节主要讲XMem是如何提升基于软件的cache优化的便携性和有效性。

为了提高cache的性能，许多软件优化的方法是在程序中静态指明当前工作集的大小，从而最大化利用cache的局部性，缺点是cache的空间大小在应用程序级是不知道的，甚至可能是动态变化的。

XMem会把优化的意图传递给硬件，比如指定哪一个应用程序的工作集需要被高度重复使用，硬件上cache就会把相应的数据优先级抬高，如果cache大小不够，会根据传达的访问模式预取后续的数据。

**5.2 Mechanism**

（1）传达关键数据集。当应用程序知道某个数据要被重复访问时，就会调用XMemLib创建一个有高使用率数据和访问模式的atom，将atom传递给硬件。

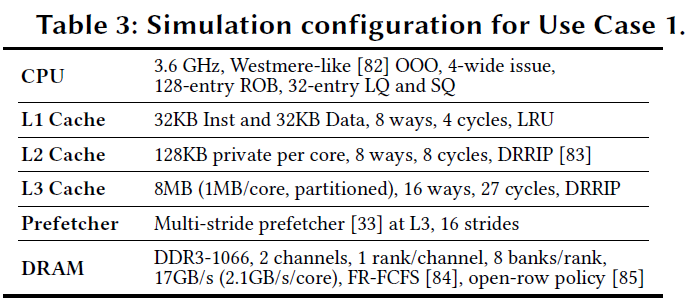
（2）优化算法。将当前活跃在core中的所有atom根据携带的数据使用率从高到低进行排序，然后按照将最高使用率的数据插入到cache，cache miss会触发预取prefetch操作。

（3）cache控制器。实施优化算法，75%的cache用来装填最高优先级的数据，其他25%的大小按照原先默认的装填方式。

（4）预取的支持。通过PAT来告诉cache应该按照怎样的访问模式预取数据。

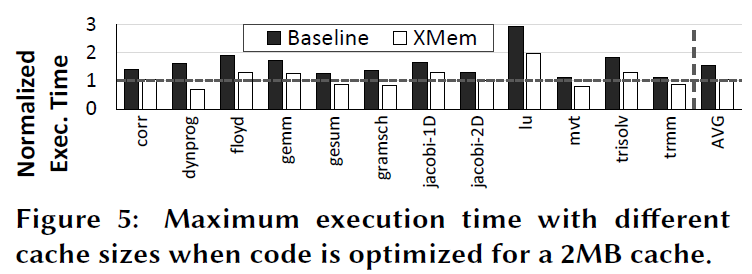
**5.3 Evaluation Methodology(方法学)**

我们按照表3中的存储层级结构模型去评估XMem系统。



**5.4 Evaluation Result**

整体性能评估。Figure 4展示的是采用不同的输入数据大小（tile大小）12个kernel的运行时间，baseline采用高性能的DRRIP cache替换算法，而XMem是使用XMem系统的性能曲线。



移植性。图5展示了缩小之前的cache大小后使用XMem与Baseline对性能的影响，发现XMem执行时间平均只增加了6%,而baseline增加了55%。

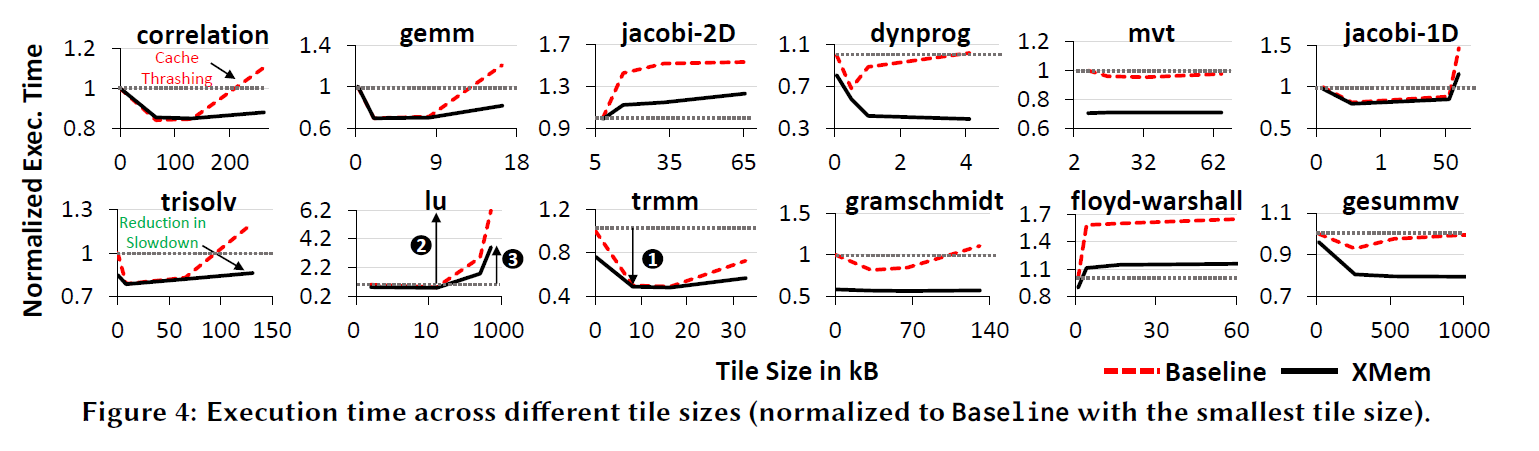
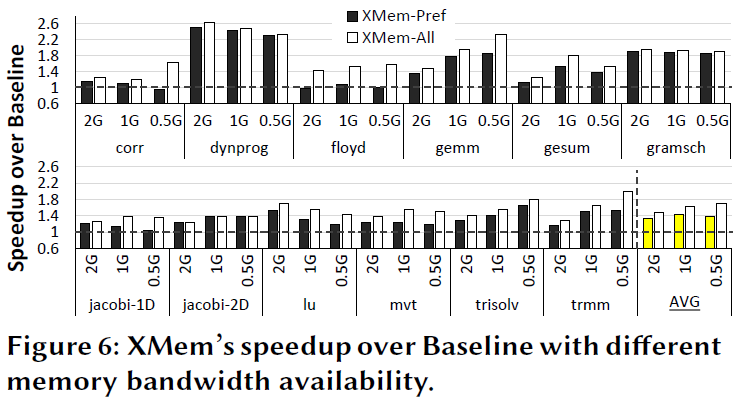


图6展示了XMem只对预取发挥作用和预取+cache管理统一发挥作用的性能进行比较。可以发现两者是系统共同提升cache的整体性能的。



**6.User Case 2：Date Placement in DRAM**

XMem通过（I）应用程序传达的访问模式语义和关键数据结构（II）系统传达的资源信息在DRAM的数据放置上更聪明的方法。

**6.1 Overview**

访存性能主要体现在访存延时上，影响访存延时的主要因素是RBL（Row Buffer Locality）和MLP（Memory Level Parallel），具体来说就是数据是如何映射在DRAM的channel、bank和row上的，由OS和存储控制器控制。

XMem能够区分不同数据结构，把不同的访问模式语义提供给OS和访存控制器，两者根据语义选择更加聪明的bank、channel和row。

**6.2 Mechanism & Algorithm**

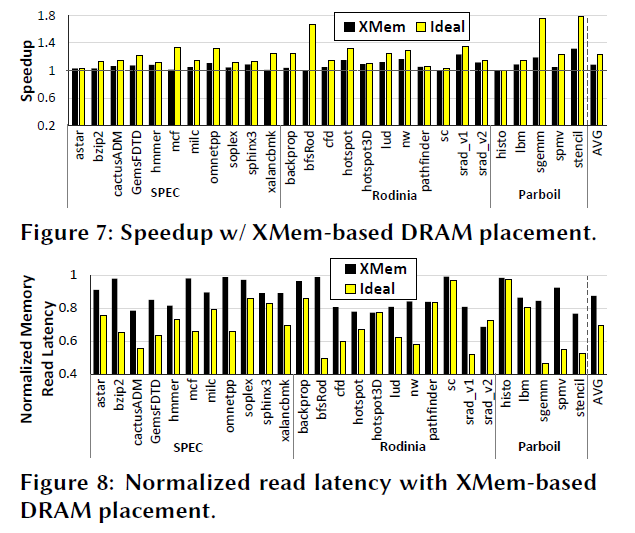
实现机制包含3个步骤。第一，在装在程序是OS通过阅读atom携带的语义获取应用程序的数据结构属性；第二，基于获得的程序语义，OS决定怎么将atom映射到DRAM的channel和bank上；第三，OS通过OS-XMem接口将atom对应物理地址处的数据放置到具体的DRAM channel和bank上.

**6.3 Evaluation Methodology(方法学)**

基于软件的存储层次结构和Table 3相同。baseline采用性能最好的性能DRAM数据放置方式，采用的数据集是SPRC CPU2006、Rodinia、Parboil等27个访存密集型的工作集。

**6.4 Evaluation Result**

Figure 7展示的是XMem和理论最理想结果相较于Baseline的性能加速比，Figure 8展示的是针对访存延时两者的降低比例。



针对图7加速比，基于XMem的DRAM放置相比于baseline策略平均提升了8.5%，而理论上限是24.4%。针对图8访存延时，读的访存延时降低了12.6%~31.4%,而写的访存延时降低了6.2%.综合起来，通过传达程序语义,XMem系统能够帮助OS使用更聪明的DRAM放置策略，从而降低访存延时、提升访存性能。

1. **Related Work**
2. XMem比基于软件的优化技术更具有通用性，不依赖于具体的编程语言；XMem比GPU中的Locality Descriptor更具通用性，因为它能传达更通用的语义。
3. 之前对访存优化主要是在软件层面，比如基于软件的预取指令、cache的旁路技术、软硬件的协同预取（使用编译器剖视后通知硬件）、程序注释将数据放在异构存储上。XMem系统与这些方法不同，第一它不指导硬件，而仅仅是向硬件传递高层语义；第二XMem系统提供的是一个通用的接口。
4. Xmem致力于向系统/硬件部件传递更具一般性的程序语义，从而在低开销下提升性能。不像PARD、Labeled RISCV这种通过给访存请求打标签实现语义传递的系统，我们设计了一个新的抽象（atom）能无缝嵌入到不同的编程语言、系统和ISA中。所以，PARD、Labeled RISCV可以看做是XMem的补充。
5. **Conclusion**

本篇文章提供了实现跨层接口的事例来弥补应用程序和微体系结构/系统之间的语义gap。我们引入了XMem系统，并使用了两个事例验证和评估。（I）静态软件cache优化，通过提供数据局部性的语义。（II）基于OS的DRAM页放置，通过区分不同数据结构和访问模式。结论是，XMem系统为我们提供一种丰富的、通用的、低开销的接口，减少了语义gap、提升了性能优化。