北京大学

硕士研究生毕业论文

题目：**虚拟化环境下基于平均淘汰时间模型的内存工作集预测**

姓　　名： 侯放

学　　号： 1401214257

院　　系： 信息科学技术学院

专　　业： 计算机系统结构

研究方向： 虚拟化

导　　师： 汪小林教授

二〇一七年五月

版权声明

摘要

虚拟化技术是云计算的基础，它对于数据中心资源的整合和高效配置起着重要的作用。内存作为一个有限的资源，如何在一个物理节点上的多个虚拟机之间动态分配来实现内存的高效利用一直是研究的难点。要实现内存的动态调配，首先要解决的问题是对虚拟机进行实时的内存预测。

在虚拟化环境下，虚拟机的内存访问对于虚拟机管理器来说是透明的，传统方法通过给虚拟机页表项置位, 来使得虚拟机的每次页面访问都发生失效并陷入到虚拟机管理器，从而获得虚拟机的内存访问序列，之后根据内存访问序列构建失效率曲线(MRC)，预测工作集大小。然而这种方法会带来巨大的开销，因为虚拟机的每次内存访问都会发生缺页中断，这对于虚拟机的使用者来说是不可接受的。

传统的通过访存序列计算 MRC 的算法开销很大，模拟 LRU 过程的链表算法的时间复杂度为 $O(NM)$(N 为访存的次数，M 为访存的不同地址的总数)，平衡树算法将时间复杂度降低到 $O(Nlog_M)$，采样的方法通过截获部分内存访问的办法减少 N 的大小，这些优化方法需要在精度和开销之间做出权衡，特别是引入采样方法后，需要分析采样对于 MRC 精度的影响。

最新的 AET 算法是基于平均淘汰时间的算法，重用时间相比重用距离能够在 O(1) 的时间获得，所以 AET 算法整体的时间复杂度降低到 O(N)，更重要的是 AET 提出的采样算法能够在只截获部分访存情况下获得精确度很高的近似 MRC，在离线情况下，使用采样的 AET 算法能够得到与传统 LRU 算法 1% 误差的结果。本文将 AET 算法引入到虚拟化下内存工作集的在线预测上来，在全虚拟化环境下实现了内存的采样截获，对比了通过采样方式使用 AET 算法获得的 MRC 和不采样使用经典 LRU 算法计算的 MRC，验证了 AET 算法在虚拟化环境下进行内存工作集预测上的可行性，通过动态采样率的方法将开销进一步降低，使得虚拟机内存预测系统能够在不影响虚拟机正常使用的情况下实现较为精准的工作集预测。

关键词：虚拟化，工作集，AET 算法，动态采样率

Memory Working Set Prediction for Virtual Machine Using AET Model

Hou Fang (Computer Science)
Directed by Prof. Wang

ABSTRACT

Virtualization technology is the basis of cloud computing, which plays an important role in the integration of resources and efficient allocation. How to dynamically allocate virtual machines' memory on one physical node to achieve full utilization has always been a challenge. To achieve dynamic allocation of memory, the first problem to be solved is the real-time memory prediction.

In virtualization environment, virtual machines' memory access is transparent to the virtual machine manager(VMM). Our work intercepts virtual machine's memory access by marking page table's reserved bit. According to the memory access trace, we can build miss ratio curve (MRC) and make predictions of the working set size. However, this kind of method can bring huge cost, because every memory access of virtual machines will cause a page fault, which is unacceptable to the user of the virtual machine.

The traditional way to compute MRC using memory access trace has high overhead. The time complexity of LRU algorithm is O(NM)(N is the number of memory access, M is the number of different accessing address). Balancing tree algorithm reduces overhead to $O(Nlog_M)$. Sampling method further cuts down the frequency of intercepting memory access meaning lower N. These optimization looks for balance between accuracy and overhead.

AET is the latest novel algorithm which reduces time complexity to O(N). More importantly, its sampling algorithm only captures a portion of memory accesses to obtain approximate MRC with high precision. This paper implements memory sampling technology in full virtualization environment. Compared the two MRCs computed by AET sampling algorithm and classical LRU algorithm, we verify the feasibility of using AET sampling algorithm in the working set prediction. Using dynamic sampling rate will further reduce the cost which makes it possible to obtain accurate working set prediction without any influence to the usage of virtual machines.

KEYWORDS: Virtulization, Working set, AET, Dynamic sampling rate

目录

第一章 序言

1.1 研究背景

随着互联网的高速发展，云计算正在被越来越多的用户使用。云计算的目标是将各种 IT 资源以服务的方式通过互联网交付给用户。计算资源、存储资源、软件开发、系统测试、系统维护和各种丰富的应用服务，都将像水和电一样方便地被使用，并可按量计费。云计算让用户能够收益于当前主流的技术而无需深入的了解和掌握他们，旨在降低成本和帮助用户专注于他们的核心业务，而不是让 IT 成为他们的阻碍。虚拟化实现了 IT 资源的逻辑抽象和统一表示，在大规模数据中心管理和解决方案交付方面发挥着巨大的作用，是支撑云计算伟大构想的最重要的技术基石。虚拟化提供了隔离多个虚拟机的能力，多个虚拟机能够共享一个物理主机并且做到相互之间的隔离。虚拟化对于资源的整合，计算机安全都起到了重要的作用。那么如何利用虚拟化技术使得有限的资源能够得到最大程度的发挥和利用成为了一个富有挑战的问题。应用的行为会发生周期性的变化，他们使用到的资源也是在时刻发生变化，于是就会存在有的虚拟机资源缺乏而有的虚拟机资源过剩的情况，内存资源就是一个典型的多个虚拟机竞争使用的资源，因此做好内存资源的优化调度对于资源的合理利用具有非常重要的意义。

1.1.1 虚拟化技术

虚拟化技术是一种将计算机的各种实体资源，如处理器，内存，网络及存储进行抽象转换后呈现出来，用户不再是使用一个物理实体，他们使用的是虚拟化技术提供出来的虚拟资源。于是多个用户能够共享一个物理资源，比如一个处理器能够同时运行两个操作系统，不同任务之间有不同地址空间，映射到不同的物理内存区域。

虚拟化的模式有很多种，图1.1是 xen 半虚拟化技术架构图，xen 是一个开放的源代码虚拟机监视器，由剑桥大学研发。xen 有两种虚拟化方式：全虚拟化和半虚拟化，全虚拟化中 DomU 中的各个硬件都是由 VMM 和 Dom0 虚拟和模拟实现，半虚拟化中 DomU 中的 CPU、Memory 由 VMM 模拟实现，IO 等设备分为前端 (Frontend) 和后端 (Backend)，前端工作在 DomU 中，而后端工作在 Dom0 中。

根据虚拟化技术的特点，我们不难发现虚拟化技术拥有很多优点：

- 提升资源利用率：通过服务器虚拟化的整合，提高了 CPU、内存、存储、网络等设备的利用率，同时保证原有服务的可用性，使其安全性及性能不受影响。

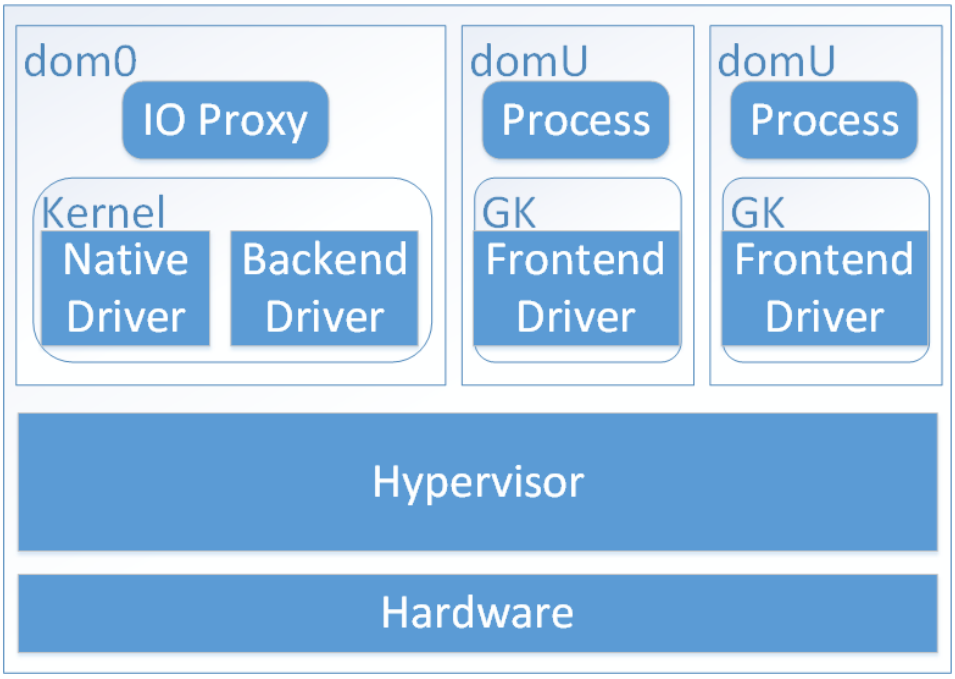


图 1.1　xen 半虚拟化架构

- 提高服务可用性：用户可以方便地备份虚拟机，在进行虚拟机动态迁移后，可以方便的恢复备份，或者在其他物理机上运行备份，大大提高了服务的可用性。
- 加速应用部署：采用服务器虚拟化技术只需输入激活配置参数、拷贝虚拟机、启动虚拟机、激活虚拟机即可完成部署，大大缩短了部署时间，免除人工干预，降低了部署成本。
- 降低运营成本：服务器虚拟化降低了 IT 基础设施的运营成本，令系统管理员摆脱了繁重的物理服务器、OS、中间件及兼容性的管理工作，减少人工干预频率，使管理更加强大、便捷。
- 降低能源消耗：通过减少运行的物理服务器数量，减少 CPU 以外各单元的耗电量，达到节能减排的目的。
- 提高应用兼容性：服务器虚拟化提供的封装性和隔离性使大量应用独立运行于各种环境中，管理人员不需频繁根据底层环境调整应用，只需构建一个应用版本并将其发布到虚拟化后的不同类型平台上即可。
- 动态调度资源：在服务器虚拟化技术中，数据中心从传统的单一服务器变成了统一的资源池，用户可以即时地调整虚拟机资源，同时数据中心管理程序和数据中心管理员可以灵活根据虚拟机内部资源使用情况灵活分配调整给虚拟机的资源。

1.1.2 内存虚拟化

计算机的内存管理使用的是页式管理，应用程序使用到的内存地址是虚拟地址，虚拟地址需要经过内存管理单元 (MMU) 的翻译转换成物理地址才能真正访问内存。其中的地址转换用到的是页表，页表保存在内存中，为了降低地址转换所带来的访存数，在处理器里又添加了转译后备缓冲器 (TLB) 加速地址转化。

而在虚拟化环境下，这样一层地址转化是不够的，虚拟机之间感受不到对方的存在，他们会认为自己独占整个物理内存空间，因此虚拟机的地址翻译可能会将同一块物理地址分配给不同的虚拟机，这就无法实现虚拟机之间的隔离和数据的安全。为了实现内存虚拟化，让客户机使用一个隔离的、从零开始且连续的内存空间，虚拟机管理器会引入一层额外的地址空间，称为客户机物理地址空间 (Guest Physical Address,GPA)。这个地址空间并不是真正的物理地址空间，只是虚拟机虚拟地址空间映射到的物理地址空间，对于虚拟机来说 GPA 是连续且从零开始的，GPA 还需要进一步的地址转换映射到真正的物理机的物理地址上。GPA 不能用作物理机内存寻址，还需要做一个转化，在全虚拟化下通常有两种方法能够做到，影子页表和 EPT 页表，他们都通过建立 GPA 到物理地址映射来实现。

内存虚拟化是实现虚拟机内存隔离的手段，为了保证内存资源的有效利用，还需要用到内存调度，当虚拟机内存不足时增加虚拟机的内存，内存富裕时让出内存，内存调度用到的技术是气球驱动技术，它是虚拟机操作系统里的一个模块，通过充气和放气来分别回收和返还内存，通过充气回收的内存交给虚拟机管理器管理供其他虚拟机使用。

1.1.3 虚拟化下动态内存调度的挑战

目前为止，虚拟化技术已经提供了一套完整的内存管理方案，包括内存资源的隔离，共享，不同虚拟机间内存的调度，但是什么时候去调度内存，不同虚拟机之间需要调度多少内存仍然是一个挑战，要做到精确内存调度的前提就是精确预测虚拟机当前的内存使用量。然而应用程序的行为是时刻变化着的，通常我们使用工作集 (Working Set Size，WSS)[4] 去刻画最近一段时间虚拟机的内存使用量。

工作集 (WSS) 定义一个进程在 t 时刻的工作集 $W(t,\tau)$ 为进程在 $(t-\tau,t)$ 时间段里访问到的内存大小。但是工作集理论有一定的缺陷，它并不直接和应用程序的性能相关。比如一个应用程序顺序扫描 100 个页面，那么在这个短暂周期里面，它的工作集大小就是 100 页，但是给这个应用程序分配一个页面或者 100 个页面的内存缺失率是一样的，这个缺失率都是 100%。

另外一个更好的方法是使用失效率曲线 (Miss Ratio Curve, MRC) 来刻画内存大小

和性能之间的关系，MRC 能够反映在不同缓存大小下的缓存失效率，这里的缓存概念是宽泛的，可以是 CPU 和主内存之间的缓存 (cache)，也可以是内存和磁盘之间的磁盘缓存 (disk cache)，当然内存也可以看作是磁盘的缓存。内存的失效意味着内存里没有当前访问的页面，这个页面可能被交换到了磁盘的交换分区，所以这次内存访问还要涉及从磁盘交换分区将这个页面换回内存的操作，我们称这个过程为 swap。

传统的计算 MRC 的办法是使用 LRU 栈算法 [13]，使用 LRU 链表去模拟缓存的写入替换，由于链表的插入移动开销都是 $O(n)$，所以算法的时间复杂度很高，对于 LRU 栈算法有平衡树优化算法能够将缓存块查询移动开销降低到 $O(logn)$。

无论是用定义去计算工作集即统计过去 t 时间里的访存数还是用 MRC 方法去估计工作集，都面临需要完整访存序列的要求，但是在虚拟化环境下获得完整访存序列的时间开销很高，上述算法并不适用于在线计算。最新的研究 AET 算法 [8] 使用基于平均淘汰时间的 MRC 计算方法，该模型用到的重用时间利用 hash 表能够在 O(1) 的时间获得，首先它降低了计算的复杂度。其次，基于平均淘汰时间的 AET 模型在使用采样的情况下仍然能够获得较高的精度。该算法目前还只用于离线数据分析，在已获得访存序列的情况下 AET 模型能够获得和传统 LRU 算法相近的精度，我们希望能够将 AET 模型用于虚拟化环境下内存精确预测，如何实现该模型以及验证该模型在内存工作集预测上的时间开销以及准确度是这篇文章的重点。

1.2 本文的主要贡献

本文实现了基于平均淘汰时间 (AET) 的内存工作集预测系统，主要贡献如下：

- 深入研究 linux 操作系统页面管理方式以及虚拟化内存管理方式，根据虚拟化内存管理原理实现了透明的内存截获方法，获得虚拟机内存访问序列。
- 根据截获到的虚拟机内存访问序列，基于平均淘汰时间模型构建重用时间分布，做到能够实时构建失效率曲线，估计虚拟机当前工作集大小。
- 为了降低开启该系统所带来的对于虚拟机的性能损失，引入热页集，降低截获虚拟机内存访问的频率，从而降低了因为截获内存访问所带来的时间开销。
- 将基于平均淘汰时间模型里的采样方法引入该系统，进一步降低截获内存访问的频率，提出动态采样率算法，根据程序运行的阶段性，动态调整采样率到合适的范围，控制系统平均开销在 2% 以内。
- 通过实验验证该系统在内存工作集预测上的可行性，准确度，以及评估引入该系统之后带来的虚拟机性能损失

1.3　本文的组织结构

- 第一章序言介绍背景知识，引入内存工作集预测的需求和难点。
- 第二章介绍相关工作，主要涉及工作集预测常用的工具——MRC 曲线，包括 MRC 曲线的计算方法和算法优化，以及 MRC 曲线在多层次存储结构下的应用。介绍已有的内存工作集预测方法，以及对于这些预测方法的优化。
- 第三章介绍 linux 下页面管理方法以及全虚拟化下内存虚拟化的方式。
- 第四章设计基于平均淘汰时间 (AET) 模型的内存工作集预测系统，包括如何在全虚拟化环境下截获客户机的内存使用，如何实现 AET 算法，以及通过各种手段降低预测系统对于虚拟机性能的影响。
- 第五章对本文实现的全虚拟化下内存工作集预测系统的正确性、可行性进行实验分析。
- 第六章总结本文的工作。

第二章　相关工作

内存工作集的预测一直是具有挑战的研究内容，无论是在裸机还是在虚拟化环境下。由于操作系统和硬件的若干优化，内存访问的模式发生很大的变化，大量的内存访问都是由硬件直接完成，软件并不知晓具体的访存行为。在不知晓访存行为的情况下，预测内存工作集的方式大致可以分为系统统计法和采样标记法，系统统计法由进程的活跃页面数等统计值直接提供，是一个进程累积一段时间的统计值；采样标记法通过随机给一组页面做记号，一段时间后扫描所有页面统计标记页面的比例从而估算总的页面使用情况。而在知晓访存行为情况下，通过计算失效率曲线 (MRC) 能够刻画内存数目和应用程序性能之间的关系。相比于系统统计法和采样标记法，这种方法具有更高的精度。

要获得访存行为能够利用的方法只有页面截获法，页面截获法通过操作页表项人为制造缺页错误，使得硬件在内存读取时发生错误陷出到缺页中断处理流程，软件修复人为的缺页错误的同时也就截获了这次内存访问。但是这种方法的开销很高，因为页面截获增加了内存访问的路径长度，所以这种方法需要优化的手段。

在本章中，我们首先会介绍失效率曲线 (MRC) 的定义，讨论计算 MRC 的经典算法，以及围绕这个经典算法做出的优化，阐述如何利用 MRC 去预测分级存储器体系结构下各级缓存的大小。然后介绍一下内存工作集预测的方法以及最新的关于内存工作集预测的优化方法。

2.1　失效率曲线 MRC

2.1.1　MRC 定义

失效率曲线 (MRC) 是衡量缓存大小和性能之间的一个重要的工具，它刻画了不同缓存大小下所对应的失效率，通过 MRC，我们可以重新定义工作集大小，即在不显著影响应用程序性能条件下所需要的缓存大小。

如图2.1所示，当缓存大小为 0 的时候，失效率为 100%，即所有的访问在缓存中均会失效，随着缓存大小的增大，失效率将会逐渐降低，这是由程序的局部性决定的，这里的局部性主要指的是时间局部性 (如果一个信息项正在被访问，那么在近期它很可能还会再次被访问)。有了 MRC，我们可以定义工作集大小为失效率为 M 时的缓存大小，M 是应用程序所能容忍的失效率。

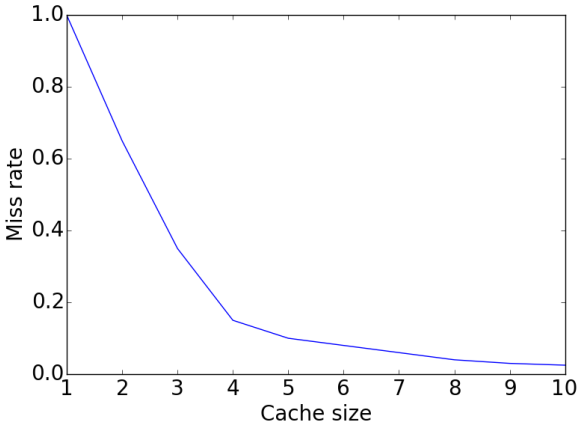


图 2.1　MRC 示例图

2.1.2　MRC 算法

传统的计算 MRC 的算法是 LRU 栈算法[13]，当我们得到一个访存序列的时候，顺序扫描这个序列，在 LRU 栈里面查找当前访问的地址所处的栈的深度，这个深度即是这个访问的重用距离，并在对应的重用距离直方图加一，完成这个操作之后将这个访问地址移动到 LRU 栈的头部表明这个访问地址是最近被访问到的，扫描完整个序列之后我们就得到了重用距离分布图。

表 2.1　LRU 栈算法示例

访问序列	a	b	a	c	b	b	c	a
栈深度	∞	∞	1	∞	2	0	1	2

表 2.2　LRU 栈算法示例

重用距离 d	0	1	2	3	≥ 4
频率 f_d	$\frac{1}{8}$	$\frac{2}{8}$	$\frac{2}{8}$	0	$\frac{3}{8}$

表2.1是 LRU 栈算法的示例，每一次访问都会引起栈中元素的移动，最近的访问会被放在栈的头部，其余元素依次向栈的尾部移动。如果这个访问地址在栈中出现过，则获得它在栈里的深度，并将这个访问地址从原来位置移动到栈的头部。初始时候栈为空，所以第一次访问时这个地址的栈深度可以理解为无穷大，当一个访问地址被再次访问时这就叫做重用，重用距离即是该地址在栈里的深度。如果将一个地址的第一次使用也看作重用，那么这次的重用距离也就理解为无穷大。于是在扫描完一次访问

序列之后我们就得到了如表2.2所示的重用距离分布表。通过表2.2，我们能够得到失效率曲线的计算公式，对于给定缓存大小 d，对应的缓存失效率 MRC_d 为：

$$MRC_d = \sum_{i=d}^{\infty} f_i \tag{2.1}$$

LRU 栈算法的时间开销不难分析，每一次的访问都首先要在栈里查找这个地址是否使用过，查询开销为 $O(n)$，如果没有找到则添加这个访问地址到栈的头部，找到则将这个访问地址从链表的当前位置移动到链表的头部，这两种情况的开销都是 $O(1)$，用数组保存重用距离分布表，那么每次更新重用距离分布表的开销也是 $O(1)$，空间开销包括保存栈的链表以及重用距离分布表，其大小均和访问的不同地址数目相关，假设访问的不同地址总和为 M，空间开销则是 $O(M)$。

虽然 LRU 算法是最为经典的 MRC 计算方法，但是其算法的时间复杂度和空间复杂度都较高，所以近来有很多的研究在 MRC 的精确度和开销上做了一些平衡，试图以较低的开销得到近似精确的 MRC。

表 2.3　Counter Stack 示例

(a,	b,	c,	a)
1	2	3	3
	1	2	3
		1	2
			1

Counter Stack[26] 创造了一个新颖的矩阵，给定一个访存序列 (a,b,c,a)，能够得到表2.3所示的矩阵, 第 i 行第 j 列表示从第 i 次访问到第 j 次访问中的不同地址的个数，利用这个矩阵，通过推导我们就能够计算出每一次访问它的重用距离。但是如果将这整个矩阵存放在内存中，其空间开销会是访存长度的平方。通过降低采样和剪枝，Counter Stack 实现了亚线性开销，并且使用采样并没有影响精度。

Shards[23] 降低开销的方式更加直观一些，对于一个访存序列来说，Shards 采用对地址进行 hash 的方式选择性地进行监控，即满足 $hash(addr)\ \%\ T\ <\ P$ 的地址才被记录下来，此时的采样率为 $\frac{T}{P}$，因为有了采样，所以实际上得到的重用距离会偏小，Shards 通过将实际得到的重用距离除以采样率 $\frac{T}{P}$ 来近似估计真实的重用距离，实验表明这样得到的 MRC 和真实 MRC 的误差也非常小。

AET[8] 跟前面研究计算访问的重用距离的方式不同，它记录的是重用时间，即同一个地址两次访问之间间隔的重用时间，这个重用时间可以用处理器的周期数来度量，

AET 利用缓存的淘汰时间进行建模，推导出了重用时间分布和 MRC 之间的量化关系。重用时间的获得比重用距离的获得要更加轻松，只需要 hash 表保存每个地址的上次访问时间便能够做到，并且每次更新重用时间分布的开销都是 $O(1)$。进一步，AET 通过采样的方式决定是否要监控访问序列中的某一次访问，不同于 Shards 对地址做 hash，采样的集合是固定的，AET 实现的采样是随机采样，每一个访问序列都有概率被监控到。因此 AET 算法的时间空间开销也都是亚线性的。

2.1.3 MRC 应用

MRC 刻画了在不同缓存大小下的访存失效率，在分级存储器体系结构中，缓存(cache)、内存、磁盘缓存、甚至磁盘都可以称之为他们下一级存储层的缓存，MRC 理论也大量应用在缓存大小的分配以及在共享缓存情况下的缓存划分策略上。

RapidMRC[21] 是一个软件定义的在线方法去刻画进程的缓存需求，他们利用现代商业处理器所提供的体系结构上的支持去计算 L2 的 MRC，并以此来优化 L2 的共享划分，这个体系结构上的支持用到的是 PMU(Performance Monitoring Units)，PMU 能够将内存访问的数据地址记录在寄存器里面。为了节省开销，RapidMRC 只截获了 L1 的数据缓存失效 (L1 data cache miss) 相关的访存地址，并且在 PMU 将地址写入寄存器时发生中断从而截获这次 L2 的访问地址，通过截获到的一系列访存序列构建 MRC。但是我们的实验发现在虚拟化环境下用中断方式去获得访存序列开销非常严重，因为这还涉及到指令执行权限从虚拟机 ring1 到宿主机 ring0 的切换。所以这种方法在虚拟化环境下并不适用。不过我们仍然使用 PMU 的计数功能去获得我们想要得到的硬件事件数，比如 TLB miss 和 Memory Reference 数。

Zhou[30] 使用了两种方式在运行时动态计算 MRC，一种方式是利用模拟器使用辅助硬件截获内存访问地址从而构建 MRC，这是一种细粒度的方式；另一种方式是使用修改操作系统的软件方式通过人为制造 page fault 截获内存访问，为了降低开销对页面进行了分组而不是以页为单位进行计算，所以这种方式是粗粒度的计算方式。但是无论是方法一还是方法二都有局限，方法一需要额外的硬件帮助，方法二只能获得非常粗略的 MRC。PATH[1] 在 Zhou 的工作之上也是利用硬件辅助提供了更加通用的页面信息，利用 PATH 提供的信息，LRU 栈以及 MRC 曲线，PATH 还实现了更为复杂内存管理策略，包括自适应的页面替换算法，改进的内存分配以及虚拟内存预取策略.CRAMM[28] 同样收集详细的内存访问序列，CRAMM 是一个基于 LRU 直方图的虚拟内存管理器，它保存了每个进程的 LRU 直方图，经过修改的 JVM 和 CRAMM 通信去获得有它自己的工作集大小 (WSS) 以及操作系统可用的内存，从而调整自己的堆大小，在不造成大量不可接受的主内存失效 (major page fault) 情况下通过减小堆大小可

以有效降低垃圾回收的频率。CRAMM 建立了一个 JAVA 应用程序工作集大小和其堆大小的正相关模型，通过监控工作集大小的变化动态调整堆大小。但是在虚拟化环境下，虚拟机的工作集大小和它被分配到的内存大小没有直接关系，并且 CRAMM 要求对操作系统的修改，在全虚拟化环境下并不适用。

现在随着 redis、memcached 等内存型缓存数据库的广泛使用，内存常常作为磁盘的缓存来加速数据库的访问。Moirai[18] 提供了一个多租户、多工作负载感知的系统帮助提高这类分布式缓存的性能，Moirai 能够使分布式缓存的管理更加轻松和合理，提高整体的性能，提供多租户的隔离以及 QoS 保证。Moirai 通过不同租户、不同工作负载的 IO 访问序列来计算不同负载的 MRC，它采用的计算方法是 Shards[23]，根据不同负载的 MRC 计算出一个最优的缓存分配策略。

Multi-cache[17] 提到在当今多层级存储器架构下，数据中心通常会使用网络存储设备来存储虚拟机的磁盘，但是当不同负载下的虚拟机同时访问网络磁盘时会产生性能上的下降，因为不同的虚拟机访问磁盘的不同位置，可能两个虚拟机的局部性都很好，但是他们交替访问磁盘，那么就会带来额外的磁盘寻道时间，所以现代数据中心会在网络存储层之前，使用 SSD 或者是 NVM 作为缓存设备加速磁盘 IO 的速度。但是这里还有一个问题，如果一个虚拟机上有着很差的数据局部性和很高的随机 IO，并且由于它频繁的 IO 访问导致占据大量的缓存空间，那么别的虚拟机能够使用到的缓存大小就会减小，作者利用各虚拟机磁盘访问序列计算 MRC，利用贪心和启发式算法合理地为各个虚拟机划分缓存大小。

总的来说，大量的研究工作已经提出使用 MRC 去改进分层级存储架构的管理，包括文件缓冲管理[10][15][31]，页面管理[1][28][30]，L2 缓存管理[16][19][20][21]。MRC 能够得到在一个特定时间段内一个进程或者是包括一组进程的工作负载亦或是一整个虚拟机的工作负载的失效率和缓存大小的关系。MRC 能够帮助我们判断一组进程他的真正的缓存需求。

离线计算 MRC 相对容易因为不会有那么多的时间和空间限制，在线计算文件系统或者是存储设备的 MRC 也相对来说容易实现，因为访问地址的截获相对来说更容易实现。但是没有硬件技术的帮助，要实现内存或者是缓存的在线 MRC 计算难度很大。前面提到的内存缓存相关工作都对开销做了很大程度的优化来取得近似的 MRC。

2.2　内存工作集预测方法

为了实现系统中的内存实时的按需分配，对内存 WSS 预测就提出了两个要求，一是要做到在线计算，这就要求计算复杂度不能太高，而要实现内存的按需分配，就是要保证在线计算达到一定的精度。前人的工作往往会在精度和开销做一定的平衡。目

前实现内存预测主要有下面几种做法。

2.2.1 系统参数法

获取 WSS 最简单直接的方法就是从操作系统中直接读取。操作系统中提供了 free 和 top 等系统工具，从中可以读取到系统当前的内存使用情况。这些系统工具从本质上都是从/proc 下读取到内存信息。Magenheimer[12] 同样也使用了操作系统内置的性能状态来获取当前 WSS，进而指导内存分配。直接获取操作系统的性能参数来估算内存 WSS 是一个直接且开销较小的方法。但是其最大的问题就是系统报告的内存大小比真实 WSS 要大，这是因为操作系统对内存的释放是“懒惰”的。在页面释放的时候，如果系统中还有空闲内存，则系统会把准备释放的页面存入 page cache 以便再次使用，而这一部分内存也会被记入系统报告的参数中。如果系统使用了 balloon 驱动[22]，那么 balloon 驱动使用的那一部分内存也会被系统记录。

2.2.2 采样标记法

精确度更高的办法就是继续深入到操作系统的内部，通过截获内存访问等方法来获取 WSS。VMware ESX[22] 使用了很巧妙的采样方法，即在每个采样周期开始时随机标记一组内存页面，一个周期以后统计这些被标记的页面占总的访问页面的比例从而推算总的页面数。这种方法的缺点是会有一定的误差，并且这种方法即便准确估算出过去一段时间使用的页面数，也不能反应页面数目和性能之间的联系，不能为回收多余页面提供参考依据。

2.2.3 页面截获法

通过绘制 MRC 同样能够为工作集大小预测提供参考，当内存资源紧缺无法满足一个进程所有页面要求的时候就需要回收内存，回收多少内存我们可以根据 MRC 提供给我们的信息来决定。假如当前进程能够在 95% 页面失效率情况下没有显著影响性能，那么我们可以回收内存到失效率为 95% 时所对应的内存数量。不过页面跟踪法在具体实现上是有难度的，由于内存访问的频繁性，获得所有的内存访问将会带来巨大的开销，并且由于计算 MRC 需要对每一次内存访问都做计算，没有经过优化的 LRU 算法的时间复杂度为 $O(M)$(M 为访问到的不同页面的个数)，这个开销经过我们实验测试是几百甚至上千倍，为了做到实时计算 MRC 而又不显著影响应用程序的性能，需要对计算 MRC 的方法进行优化。前面也提到过通常有两种方式实现内存页面的跟踪，硬件方式和软件方式。硬件方式是通过在内存总线上添加小的硬件设施获得每一次的内存访问，软件方式则是人为的制造 page fault 或者是扫描页表项对页表项加标记。

2.3　内存工作集预测方法优化

表5.2总结了各种方法的特点。从这个总结中可以看出系统参数法虽然开销低，但是精确度不高，因为操作系统只能给我们有限的统计数据，而且这些统计数据是累加值，并不是过去一段时间里的值，因此不能作为实时内存预测方法。而 VMWare 提出的采样方法类似于生物学统计物种数量的方法，通过标记一部分页面，然后在回收时候根据回收页面中标记页面的比例估计总的页面使用数，这种方法具有理论基础，精确度较高，因为需要标记页面和统计标记页面的比例，所以有一定的开销，但是这种方法和用工作集定义去预测工作集方法一样无法建立内存大小和性能之间的关系。所以最近的研究成果都是试图通过页面截获的方式计算失效率曲线 (MRC) 来预测工作集，由于没有特殊的硬件辅助支持，所以通常都是用软件的方式截获内存访问。

表 2.4　内存工作集获取方式比较

项目	系统参数	采样标记	页面截获	
			硬件	软件
精确度	低	较高	高	高
开销	低	中等	较高	高
缺点	系统统计内存数偏高	无法建立内存大小和性能之间的模型	额外的硬件支持	开销高，需要优化手段

不过软件手段还需要优化才能够满足低开销的要求。经典的使用 LRU 算法计算 MRC 的时间复杂度为 $O(MN)$，其中 N 是访问序列的长度，M 是访问到的不同地址的个数，空间复杂度为 $O(M)$，Bennett[2] 使用线段树来优化算法，将时间复杂度降低到 $O(NlogM)$，足迹理论 (footprint theory)[5][27] 是最早能以线性时间计算 MRC 的算法，该方法采用了重用时间的概念而非重用距离，其时间复杂度为 $O(NloglogM)$，利用足迹理论得到的 MRC 具有非常高的准确性并且这种方法已经被多个应用所采用[24][6][7][11]。而 Shards[23] 和 Counter Stack[26] 则是首次提出降低空间开销的技术，并且他们的时间复杂度也是线性的。除了优化计算 MRC 算法，别的降低开销的软件方法也被提出。比如利用程序的局部性原理将内存分为活跃内存 (active) 和不活跃内存 (inactive)(有的文章里称为 hot 和 cold)[30][28][29]，对于活跃内存我们不去截获他们的访问，只截获不活跃内存的访问，计算不活跃内存的 MRC，最后再加上活跃内存的数量即可估计总的内存使用量。还有文章利用程序的阶段性，如根据硬件计数器的变化情况来推测当前内存使用量是否在发生剧烈的变动，如 TLB miss 的值就能够反应当前应用使用内存情况的

波动，当应用程序的内存使用波动不剧烈的时候，可以选择不截获内存[29][21][3]，这些方法对降低开销都起到了一定的作用。

第三章　虚拟化下的内存管理

前面提到内存预测的相关工作都是采用截获缺页异常 (page fault) 的方式来获得内存访问序列。本章我们将要简要回顾一下操作系统基于页的内存管理方式，以及在虚拟化下内存是如何做到在多个虚拟机之间共享，隔离，不冲突地使用物理内存。本章是下一章实现虚拟化下内存工作集预测的理论基础，我们将在现有的内存虚拟化下实现扩展以达到我们的目的。

3.1　页式管理

在现代操作系统中，各进程的虚拟空间划分成若干个长度相等的页 (page)，而物理内存划分为同样大小的页框 (page frame)，程序加载时，可将任意一页放人内存中任意一个页框，这些页框不必连续，从而实现了离散分配。该方法需要 CPU 的硬件支持，来实现虚拟地址和物理地址之间的映射。在页式存储管理方式中虚拟地址由两部分构成，前一部分是页号，后一部分为页内偏移。

如何通过页号找到真正的物理页框，并且保证在发生进程切换后不同进程的地址空间能够映射到正确的物理地址上，就需要为每个进程引入新的数据结构，这个数据结构就是页表 (page table)。在进程加载时，页表也同时被加载进内存。不难计算要维护虚拟地址中的页号到物理页号的映射，以 32 位操作系统和 4K 页为例，页内偏移占 12 位，剩余 20 位表示页号，那么一共 2^{20} 个页，需要 4MB 的空间，这样显然浪费空间，所以页表的管理都是采用分级的页表。

64 位系统下采用的是如图3.1的地址转换，这个转换过程通常是由处理器的硬件直接完成，不需要软件参与，这个硬件称为内存管理单元 (MMU)。通常，操作系统只需在进程切换时，把进程页表的首地址装入处理器特定的寄存器中 (cr3)。一般来说，页表存储在主存之中。这样处理器每访问一个在内存中的操作数，就要多次访问内存。

- 读取内存中的页表完成逻辑地址到物理地址的转换，由于多级页表的存在，读取内存的次数还不止一次
- 使用地址转换得到的物理地址完成真正的内存读写操作。

这样做时间上耗费严重。为缩短查找时间，可以将页表从内存装入 CPU 内部的关联存储器 (例如，快表 TLB) 中。此时的地址转换过程是：在 CPU 给出有效地址后，由 MMU 自动将页号送人快表，并将此页号与快表中的所有页号进行比较，而且这种比较是同时进行的。若其中有与此相匹配的页号，表示要访问的页的页表项在快表中。

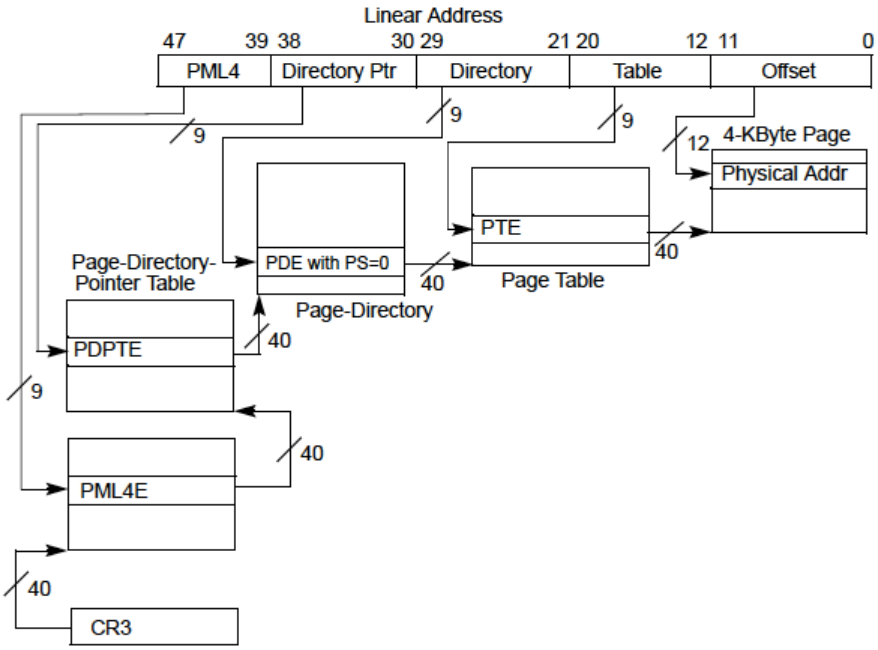


图 3.1　地址翻译

于是可直接读出该页所对应的物理页号，这样就无需访问内存中的页表。由于 TLB 的访问速度比内存的访问速度快得多，这就加速了地址转换的过程。

在程序的执行过程中，因为遇到某种障碍而使 CPU 无法最终访问到相应的物理内存单元，即无法完成从虚拟地址到物理地址映射的时候，CPU 会产生一次缺页异常 (page fault)，从而进行相应的 page fault 处理。基于 CPU 的这一特性，Linux 采用了请求调页（Demand Paging）和写时复制（Copy On Write）的技术。

- 请求调页是一种动态内存分配技术，它把页框的分配推迟到不能再推迟为止。这种技术的动机是：进程开始运行的时候并不访问地址空间中的全部内容。事实上，有一部分地址也许永远也不会被进程所使用。程序的局部性原理也保证了在程序执行的每个阶段，真正使用的进程页只有一小部分，对于临时用不到的页，其所在的页框可以由其它进程使用。因此，请求分页技术增加了系统中的空闲页框的平均数，使内存得到了很好的利用。从另外一个角度来看，在不改变内存大小的情况下，请求分页能够提高系统的吞吐量。当进程要访问的页不在内存中的时候，就通过 page fault 处理将所需页调入内存中。
- 写时复制主要应用于系统调用 fork，父子进程以只读方式共享页框，当其中之一要修改页框时，内核才通过 page fault 处理程序分配一个新的页框，并将页框标记为可写。这种处理方式能够较大的提高系统的性能，这和 Linux 创建进程的操作过程有一定的关系。在一般情况下，子进程被创建以后会马上通过系统调用

execve 将一个可执行程序的映象装载进内存中，此时会重新分配子进程的页框。那么，如果 fork 的时候就对页框进行复制的话，显然是很不合适的。

无论是请求调页技术还是写时复制技术，他们都说明物理页框的分配是在应用程序真正使用内存的时候才会去做，那么在 page fault 处理程序中我们就能够得到页的首次访问。

前面我们提到了页面的分配，操作系统同时还承担着回收页面的责任，应用程序在需要内存的时候会向系统申请内存，但是他们未必会在使用完这些内存之后主动将页面还给操作系统，长此以往，所有进程都不交还申请的页面，只会导致需要页面的进程得不到内存，而空闲的页面也得不到释放，所以操作系统还需要提供相应的功能去回收他们。

linux 里的页面回收算法也是基于 LRU 算法的，已经很久没有访问到的页面是适合被回收和释放的，操作系统维护了两个双向链表：active 链表和 inactive 链表，这两个链表是 linux 完成页面回收重要的依据。顾名思义，经常被访问的页面也会被放在 active 链表里，不常访问的页面则被放在 inactive 链表里，页面在两个链表间移动，当然操作系统并不是简单的从一个页面一次访问就从 inactive 链表放入 active 链表，linux 还引入了两个标记位：PG_active 和 PG_referenced。一个页面初始时两个标记位都为 0。如果访问到一个 PG_active 的页则保持其在 active 链表的位置，否则，如果访问到一个 PG_referenced 为 0 的页，则将 PG_referenced 置 1 但仍位于 inactive 链表，如果 PG_referenced 为 1，则清除 PG_referenced 位，置上 PG_active 位，并将页面从 inactive 链表移动到 active 链表。页面回收发生时就会从 inactive 链表里选择尾部的页面。

3.2　虚拟化下的内存管理

为了实现内存虚拟化，让客户机使用一个隔离的、从零开始且具有连续的内存空间，内存虚拟化引入一层新的地址空间，即客户机物理地址空间 (Guest Physical Address, GPA)，这个地址空间并不是真正的物理地址空间，它只是宿主机虚拟地址空间在客户机物理地址空间的一个映射。对客户机来说，客户机物理地址空间都是从零开始的连续地址空间，但对于宿主机来说，客户机的物理地址空间并不一定是连续的，客户机物理地址空间有可能映射在若干个不连续的宿主机地址区间，如图3.2所示：

客户机的物理地址并不能直接用于内存的地址寻址，为了完成地址转换，xen 用了两种表：P2M 和 M2P，P 指的是客户机的物理地址，M 指的是机器的物理地址，每个客户机的物理地址映射到机器地址的方式是不同的，所以每个虚拟机都有一个 P2M 表，M2P 是机器地址到客户机物理地址的映射，由 Xen 维护。虚拟化有两种模式，半虚拟化和全虚拟化。半虚拟化需要修改客户机操作系统以支持底层的虚拟化设施，全

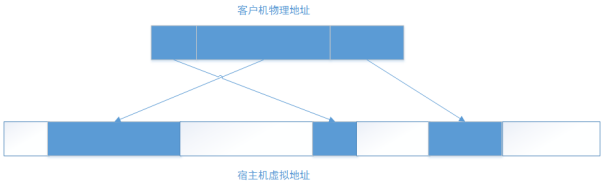


图 3.2　客户机物理地址到宿主机虚拟地址的转换

虚拟化顾名思义无需修改客户机操作系统即可移植到虚拟化环境下。我们这里研究对象是全虚拟化。

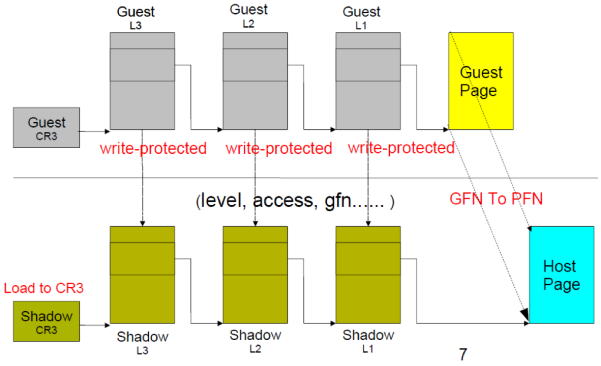


图 3.3　影子页表示例

全虚拟化下的内存虚拟化有两种方式。影子页表 (shadow page table) 和硬件辅助虚拟化 (hap)。

- 影子页表 (shadow page table)

  影子页表如图3.3所示，虚拟机管理器为每个虚拟机都维护一套影子页表，影子页表顾名思义和客户机操作系统的页表类似都是分级的页表，并且客户机页表的每一项在影子页表中都有着复制。客户机页表的起始地址为 Cr3，这个 Cr3 地址是由客户机操作系统分配，但是当客户机操作系统发出一个加载 Cr3 指令的时候，由于这是一个特权操作，而客户机操作系统运行在 ring1，它无权使用这个指令，所以运行的控制权交给了虚拟机管理器 (运行在 ring0)。虚拟机管理将 Cr3 寄存器的值写成影子页表的起始地址，这样，客户机管理系统以为它自己使用的是自己的页表，而其实是使用的影子页表。

  在使用影子页表的时候会有两种情况会发生缺页中断，一种是客户机操作系统本身的缺页中断，比如客户机操作系统还没有分配这个页，所以页表项的 present 位为 0。另外一种情况是客户机操作系统的页表和影子页表不同步。这是因为影子页表的建立是滞后的，在客户机操作系统因为访问了无效的影子页表而发生缺页中断的时候，影子页表才有了机会去更新自己的影子页表项。

当缺页发生时，虚拟机管理器首先检查引起异常的原因，然后对发生异常的客户机虚拟地址在客户机页表中所对应的执行权限进行检查，根据异常错误码确定此异常的原因，如果该异常是由客户机引起，则虚拟机管理器将权限交还给客户机操作系统。如果该异常是由于影子页表和客户机页表不同步导致，则虚拟机管理器根据客户机页表建立相应的影子页表，将真正的物理地址填充到影子页表项中，并且根据客户机页表填充影子页表相应的访问权限位。

- 硬件辅助虚拟化 (hap)

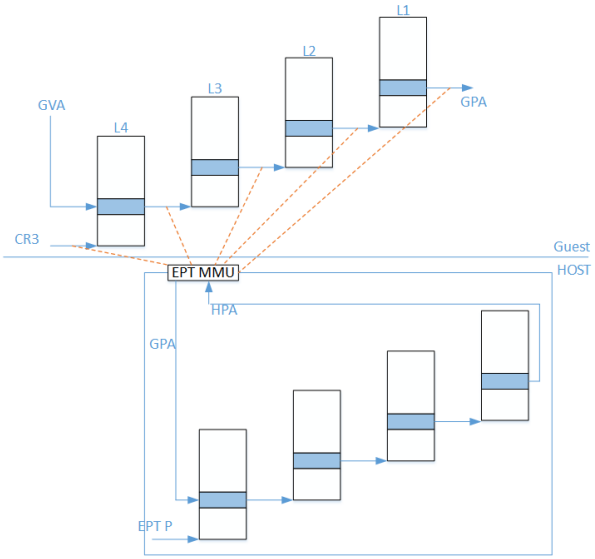


图 3.4　ept 页表示例

硬件辅助虚拟化是使用硬件的方式引入了另一个页表叫做 ept 页表，ept 页表的分级映射方式和普通的页表分级方式完全相同，如图3.4所示是使用 ept 页表进行地址转换的过程，ept 页表使用专门的 eptp 寄存器保存 ept 页表的起始位置，在客户机操作系统完成正常的页表查询得到物理地址之后，硬件自动将客户机物理地址再在 ept 页表中查询以得到宿主机的物理地址。

在客户机进行地址翻译的时候由于缺页和写权限不足等因素导致客户机退出，产生 ept 异常，对于 ept 缺页异常，虚拟机管理器将客户机物理地址映射到虚拟机管理器的虚拟地址，为虚拟地址分配新的页面，然后再更新 ept 页表，建立起客户机物理地址到宿主机物理地址映射。对 ept 写权限引起的异常，虚拟机管理器则通过更新相应的 ept 页表来解决。

我们实验使用的环境是用的影子页表，ept 模式下的实现是类似的，对于影子页表的同步还有几点需要考虑

- Access 和 Dirty 位

由影子页表的性质我们能够发现影子页表的建立一定不早于客户机页表的建立，所以客户机页表的第一次访问一定会被宿主机截获因为此时访问的影子页表不存在，这个时候在影子页表中建立线性地址到物理地址的对应关系，并将客户页表中相应项的 Access 位置 1.

对于 Dirty 位，宿主机在影子页表中建立线性地址到物理地址间的映射之初，可以将页置为只读，这样当客户机对该页进行写操作时，将由于权限不足导致页面故障，使宿主机获得控制，进而有机会确保影子页表项和客户机页表项在 Dirty 位上的一致。

- 影子页表性能

  影子页表的实现是影响客户机性能的关键因素，其最大的开销为当客户机操作系统发生进程切换而切换 Cr3 或者刷新 TLB 的时候，原来的影子页表就会被删掉重建，而客户机在实际运行过程中切换 Cr3 是非常频繁的，影子页表删掉重建的开销不容忽视。而实际上，刚被删掉的页表又很有可能马上会被使用，如果在客户机进程切换的时候，宿主机知道哪些影子页表不会被更新而保留下来，只更新会发生改变的页表，那么就能大大地降低开销。

  一种优化方法是在影子页表第一次被分配时，就把该影子页表的上一级页表项中的页表的访问权限设置为只读，这使得客户机下次要修改该客户机页表时就会发生缺页中断，但是因为一个客户机页表中可能对应多个影子页表，宿主机要保证在这时所有的影子页表的权限都是只读的。

第四章　基于平均淘汰时间模型的内存工作集预测系统设计

要完成基于平均淘汰时间模型的内存工作集预测系统，首先需要在虚拟机管理器截获虚拟机的内存访问，由于虚拟机的内存访问对于虚拟机管理器来说是透明的，所以需要额外的机制帮助虚拟机管理器。获得访存序列之后用 AET 算法计算 MRC，在计算 MRC 过程中我们发现如果不加任何优化，这样的预测系统将会给虚拟机带来巨大的开销，通过大量的实验和观察，我们设计和实现了热页集和动态采样率优化，本章将会详细介绍基于平均淘汰时间模型的内存工作集预测系统是如何设计的。

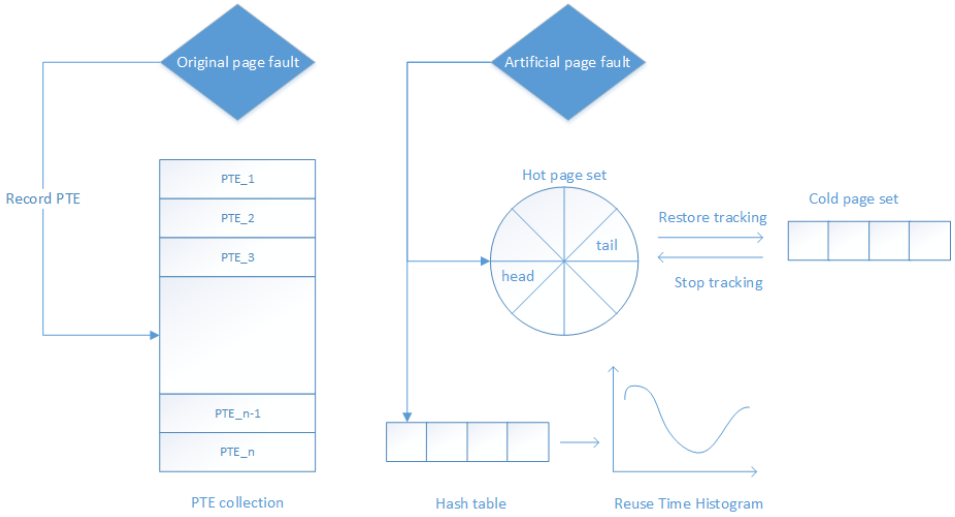


图 4.1　系统组成

图4.1是虚拟化下内存工作集预测系统的概要图，主要分为了两个部分，一部分是左半部分的页表项集合 (PTE Collection)，上一章已经分析了影子页表的更新和同步机制，通过对影子页表更新和同步的截获，能够获得客户机使用的页面的页表项，保存页表项集合之后我们能够通过对客户机页面页表项进行置位来使客户机访问页面时失效从而截获客户机页面的访问，这是系统的第一步。

右半部分和 AET 实现以及系统优化相关。AET 的计算复杂度不高为 O(1)，但是缺页中断的代价很高，为了降低整体系统的开销，就要减少缺页中断的次数，但是减少缺页中断次数又会影响到 AET 算法的精度，在经过试验权衡之后，系统通过引入热页集和动态采样率方法将开销降低到能够容忍的范围，并且还能够得到精确度很高的 MRC。

热页集 (Hot Page Set) 是将最近访问的页面加入热页集，热页集里的页面不再跟踪，即清除对该页面页表项的置位。热页集的构造是环形队列，后文会解释为什么要构造这样一个数据结构，当热页集满时，将队列尾部的页面剔除出热页集，加入冷页集，冷页集是个抽象的概念，实际并没有相应的数据结构，冷的页面我们会置位以恢复对它的跟踪。动态采样率是根据运行时的参数通过外围控制器控制采样比例，每次扫描页表项置位的时候根据采样比例对部分页面置位。

图中的 Hash Table 用来记录页面的访问时间，这样当重用发生时从 hash 表读取上次访问时间，得到重用时间，更新重用时间分布图 (Reuse Time Histogram)，更新 hash 表里的访问时间。

本章的后续部分我们将深入系统的各个部分，解释设计和实现的细节。

4.1 虚拟化环境下的内存截获

在虚拟化环境下，宿主机对于客户机的运行知之甚少，只有当运行在 ring1 的客户机执行特权指令的时候控制权限才回到宿主机 (ring0) 上，我们作为虚拟机物理资源的管理者，只有当访问权限陷入到宿主机的时候才能做一些额外的操作。要做到虚拟机内存的预测，无论是使用哪种算法，我们首先都需要获得虚拟机的内存访问序列。在页式管理的操作系统中，内存访问序列也可以看做是页面访问序列，相比于地址访问序列，页面访问序列会少很多，不仅仅是因为一个页面包含 512 个地址 (64bit) 或者 1024 个地址 (32bit)，而且由于空间局部性，在短时间内，访问的地址很可能是相邻或邻近的地址。以页面为粒度去监控内存既不失精确度还能提高效率。

而截获页面我们现在唯一能够知道的入口就是缺页中断 (page fault)，在裸机上，缺页中断发生在页面权限不够、当前使用的页还没有加载进内存等一系列原因，在虚拟化环境下，宿主机的缺页中断还发生在影子页表不存在、影子页表没有同步，影子页表写保护等。页表正确建立好之后的内存访问就回归正常，不再发生 page fault。所以我们需要人为地制造缺页中断，使得页面的每次访问都会发生 page fault 陷出到虚拟机管理器，这样我们才能够拿到完整的页面访问序列。

图4.2出自于 Intel 手册[9]，表示了各级页表目录每一个 bit 的含义，包括权限位，access 位、dirty 位等等，其中有几位需要我们特别关心，51-M 位，这几位是保留位，在手册里明确提到这几位保留位必须为 0，否则在访问的时候就会发生缺页中断。我们在实现中就是利用到这几位来人为地制造缺页中断。具体的实现分为了下述的三个步骤：

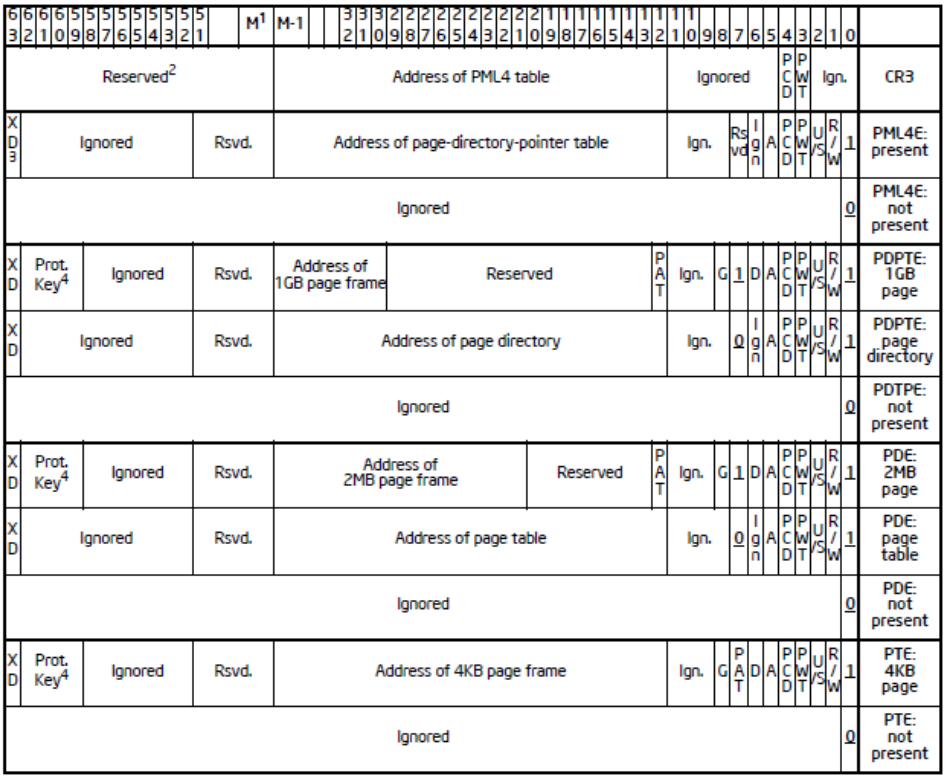


图 4.2　页表项保留位

4.1.1　PTE 收集

由于影子页表初始状况下为空，或者是在进程切换之后处于写保护状态，所以客户机进程对页面的首次访问一定会发生缺页中断，在缺页中断处理流程中，通过扫描各级页表，最终我们能够得到最末一级页表也就是页表项 (page table entry, PTE)，页表项里保存的就是客户机虚拟地址所对应的物理地址。根据缺页中断的错误码修复了相应错误之后，我们就将 PTE 对应的 51-M 保留位全部置 1，这样下一次对该页的访问就会再次发生 page fault。

但是这么做会带来一个问题，如果在这一次缺页中断里面就设置了保留位，那么当缺页中断处理完毕并将指令执行权限交还给客户机之后，客户机一执行这条访存指令就又会发生我们人为制造的缺页中断，这就导致程序死循环，该页面的缺页中断永远无法修复。简单的办法就是让缺页中断处理程序检查保留位，因为只有我们才会去置保留位，当发现是我们人为制造的 page fault 后就不再置保留位，而是将保留位全部清零保证客户机能够正常执行下去。这样做之后我们就失去了对该页的监控，我们的做法是当虚拟机下一次陷出的时候置位之前缺页中断里的 PTE，所以这就至少需要一个单位的空间来保存上次缺页中断时候的 PTE。

实际上我们并不仅仅保存上一次缺页中断的 PTE，出于后面系统扩展性的考虑，

我们将所有的 PTE 全都保存了下来，由于缺页中断里获得的 PTE 是会有重复的，所以这里我们实现集合来保证所有 PTE 没有重复。

4.1.2 置位

刚才也提到置位的时机并不是这个页发生缺页中断的时候，否则会导致死循环，而是把置位的时机放在了下一个缺页中断的时候。这样显然会带来一个问题，因为我们不是在一个页修复错误之后马上对其监控，那么在下一个页发生缺页错误之前对这个页的访问全都无法截获，举个例子，当有如下的页面访问序列：“aaaaabbbbbccccccaaaaa”，在 a 第一次访问发生缺页中断之后我们便失去了对 a 的跟踪，当第一个 b 页面访问时，才会去置页面 a，所以我们得到的访存序列为"abca"。其实这个问题并不会影响我们的结果，反而能够提升我们系统的效率，一个页内部的连续访问我们可以理解为这个页只访问了一次，因为我们的监控粒度是页而不是字节，并且一个页的连续访问也不会影响操作系统的页面替换算法。事实上在之后的一个小节中我们还将进一步把这个置位间隔设置得更大，短时间内不监控的页面更多，这部分页面我们称之为热页，下文将详细介绍相关概念。

4.1.3 修复人为错误

修复人为错误相对比较容易，只需要在缺页处理流程中判断是否是我们人为制造的错误，如果是则将对应相关的保留位清零，然后返回客户机操作系统即可。

至此我们已经能够在全虚拟化下截获到客户机页面的访问，获得页面访问序列之后就能够通过特定算法计算内存工作集，下面一节将会解释如何将 AET 模型应用在系统里。不过在运行时用上述方法截获所有内存访问带来了巨大开销，在虚拟机的性能影响非常显著，后文会通过三种方法对内存截获进行优化。

4.2 AET 模型的引入

4.2.1 AET 模型的原理

缓存系统使用的替换算法通常是 LRU 算法，根据 LRU 算法，无论缓存是如何组织的，缓存命中或者是缓存失效都会导致缓存块的移动。AET 模型和缓存块的平均淘汰时间相关，缓存块在被淘汰之前也许会发生多次重用，而淘汰时间是在缓存块最后一次被访问到至被淘汰的时间。

AET 模型基于缓存块移动的概率，假定 AET(c) 是大小为 c 的全相连 LRU 缓存的所有数据淘汰的平均淘汰时间，T_m 为缓存块移动到位置 m 的时间，很显然 $T_0 = 0$ 以

及 $AET(c) = T_c$。假定 rt(d) 为重用时间为 d 的访问次数，n 为所有的访问次数，用 f(t) 表示重用时间为 d 所占比例，则 $f(t) = \frac{t}{n}$。使用 P(t) 表示重用时间大于或者等于 t 的比例，那么 $P(t) = \sum_{x \geq t} f(x)$。缓存块在缓存中是否移动依据于概率 P(t)，这是因为如果一个缓存块所在位置为 m，那么如果下一个数据的访问它的重用时间超过了 T_m，即下一个访问的缓存块的位置在该缓存块位于 LRU 位置的尾部，那么下一个访问就会引起该缓存块的移动。而重用时间超过 T_m 的概率为 $P(T_m)$。换言之在 m 位置的缓存块向 LRU 尾部的移动速度为 $P(T_m)$，想象 LRU 是条直线道路，所有缓存块从 LRU 头部走到 LRU 的尾部，在每一个时刻都对应了一个速度，对每一时刻的速度进行积分就能得到整个 LRU 长度。

给定一个大小为 c 的缓存，根据积分公式，如果我们知道了重用时间的分布 P(t) 我们就能得到失效率曲线，$MRC(c) = P(AET(c))$，因为大小为 c 的缓存其平均淘汰时间为 AET(c)，如果重用时间超过了平均淘汰时间那么这个缓存块就很可能已经被淘汰出缓存了，所以这次重用也就引起了一次缓存失效。换言之如果我们得到了重用时间分布，MRC 曲线将很容易通过公式4.1计算获得。

$$\int_0^{AET(c)} v(t)dt = \int_0^{AET(c)} P(t)dt = c \tag{4.1}$$

4.2.2　AET 模型的实现

AET 模型需要记录的是重用时间的分布，因为是概率模型，所以也不需要全部完整的重用时间分布，这让采样计算有了可能。我们现在这个系统已经实现了页面置位，页面中断截获。而要引入 AET 主要就需要记录一下我们每次截获的页面是否是重用的页面，如果是我们需要计算重用的时间。

重用时间我们使用的是逻辑时间而非物理时间，对于一个访存序列，一个逻辑时间即是一次访存，所以重用时间就是在一个地址的重用间隔内的别的内存访问总数。重用时间的记录和计算都是非常容易实现的，用一个全局时间记录当前时间，每次访存给全局时间加一。用 hash 表根据访存地址找到上次访问的访问时间，用现在的全局时间减去上次时间得到重用时间，更新重用时间分布直方图，更新 hash 表里的该地址访问时间为当前全局时间。

在我们系统的最初实现中，我们想要获得的是每个页的重用，但是我们系统在修复页面之后，如果接下来的若干访问都是该页内的访问而没有访问别的页，我们并没有机会去给该页置位也没有办法截获到这些页内的连续访问，一旦访问了另外的页才能恢复对该页的监控。为了获得在一个页内部的连续访问次数，也就是重用时间都为 1 的这些访问数，我们想到了用硬件计数器，硬件计数器能够记录系统访问内存的次

数。但是在我们加入硬件计数器之后，我们仍然没有得到满意的结果，我们分析一个页内的访问可能读取也可能不读取内存而直接被缓存命中，如果连续访问同一个地址，那么虽然访问了多次页面，但其实都是从缓存直接读走了数据，硬件计数器不会计数。后来我们又想办法通过别的硬件数据估算重用时间为 1 的访问，最后都没有得到满意的结果。

Zhao[29] 和 Wang[25] 在构建重用距离的时候也不是拿原本系统的访存序列去计算，它们利用热页集过滤后得到的访存序列构建的 LRU 栈，之后再计算该访存序列工作集大小，由于只是添加了一个热页集缓存，并没有减少真正的访存工作集大小，就好比在现在硬件缓存基础上增加了一个人为制造的软件缓存，他的目的是利用局部性控制开销，重用距离小的频繁的访问没有必要去计算，因为它们对于整体工作集评估影响不大。我们接下来提到的系统优化中用到的热页集想法就是出自于这两篇文章，用 AET 算法估计经过热页集过滤后的访存序列。

4.3 系统的优化

4.3.1 热页集

收集了所有 PTE 之后，我们能够通过置保留位的方式来截获所有页面访问，一旦客户机访问到了被置位的页面，便会发生缺页中断陷入到虚拟机管理器，等到保留位被清零才把指令执行权限交还给客户机操作系统。但是如果所有的客户机页面访问都被截获，那么客户机基本上无法正常运行，因为发生缺页中断的开销太大，一次缺页中断就意味着一次 cache miss、TLB miss 以及上下文切换。我们统计过 SPEC 测试程序，每秒钟就会用百万次的内存访问，所以这样的做法显然在性能上无法接受。Zhao[29] 最早提出了热页集的概念，他们的做法是将页面划分成两个部分，热页和冷页，热页指的是最近访问过的页面，冷页则指的是很久没有访问过的页面，热页和冷页的概念也是来源于程序的局部性，热页是程序短时间内经常会访问的页面。热页集顾名思义就是一个包含热页的集合，最近被访问的页面被加入热页集，当然也会从热页集里面淘汰，那些许久在热页集里的页面就会从热页集剔除降级为冷页。

热页集给我们的思考是我们能不能只监控那些许久没有被访问的页面 (冷页)，而无需监控热页？答案是肯定的，热页集里页面的访问序列我们并不关心，通常热页集也不会设置很大 (Zhao[29] 把热页集的大小最大设置到了 4096 个页)，热页集作用可以类比 TLB 的作用，在 TLB 里的页表项无需访问内存就能完成地址转换，热页集的做法相当于在硬件 TLB 的基础上又设计了一层软件 TLB，好比二级 TLB，由于热页集里的页面不会被截获，所以没有办法使用 LRU 的算法管理热页集，只能采用先入先出的

原则，即先进入热页集的页会被先剔除。而我们得到的访存序列是经过这样一个软件 TLB 过滤的冷页访问序列。确实热页集对于预测“缓存”，即失效率比较大时候的内存大小会有影响，因为热页集不是采用 LRU 的替换算法，这就导致可能最近访问的页面因为更早加入热页集而被剔除出热页集，但是这只会影响重用距离直方图中重用距离小的前半部分，比如热页集大小为 2 时候如果有如下访问序列：“abacd”，经过先入先出热页集过滤得到的访存序列为“ab”，而 LRU 缓存过滤得到的访存序列是“ba”，但是我们能够发现重用距离的误差小于热页集的大小，但是截获到的访存数量不变。并且只会影响重用距离直方图中重用距离小于热页集大小的分布，对于内存工作集预测来说，我们往往只关心失效率比较小的时候的精确度，也就是说重用距离直方图中长距离的重用的精确度更加重要。

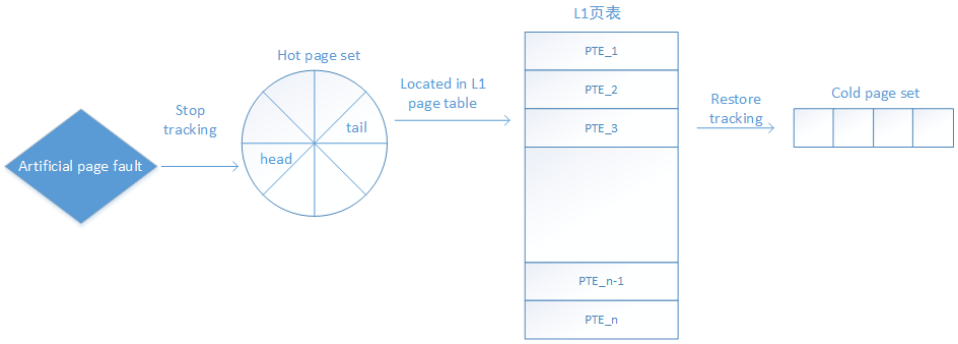


图 4.3　热页集示意图

有了上面的分析，我们知道热页集应该是一个先入先出的队列，在实现上我们使用了 FIFO 的环形队列如图4.3。当一个页面被截获到时，它就会被标记为热页加入热页集，它的相关信息被写入队列的尾部。当队列已满，队列头部的页面就会降级为冷页，并且需要置保留位恢复跟踪，接下来对该页的访问会发生 page fault，我们在热页集里面会保存该页 PTE 所在的 L1 页表的页框号以及这个页在 L1 页表中的偏移，这是为了方便从热页剔除后快速定位该页的 PTE，只需要将 L1 页表的物理页框号映射到宿主机虚拟地址再加上该页表项在 L1 页表里的偏移，就得到了 PTE 的地址。

4.3.2　采样

实验表明仅仅使用热页集仍然有很大的开销（SPEC CPU 2006 使用 2048 热页集平均开销为 557%），还需要别的优化手段降低开销。前人的工作中，Shards[23] 和 AET[8] 都提到了采样的方式降低开销，采样的方式比较直接，他们的核心观点围绕是否能够通过采样减少截获的页面数，而又能得到近似精准的 MRC 曲线。虽然对于计算 MRC 的算法有很多的优化，比如用线段树优化 LRU 栈算法，或者是发明了很新奇的低开

表 4.1　缺页中断比例

benchmark	page fault 数目	原本 page fault 数目	比例
milc	1886834662	19959422	94.533
cactus	110191655	1426631	77.239
soplex	1.573788609	459321	1249.210
gems	1731257708	555882	3114.433
mcf	1606761141	970739	1655.193
astar	2670573507	536086	4981.613
bzip2	4785138	1901917	2.515
bwaves	536873102	3045885	176.261

销的算法计算 MRC，采样是能够在根本上降低监控的页面数目。特别是当使用缺页中断的方式截获页面访问，其主要的开销是缺页中断所带来的开销而不是算法本身的开销。我们做了一个实验来验证开启我们的监控和不开启我们的监控缺页中断次数的比较。表格4.1显示开启监控之后缺页中断的次数是原本次数的百倍，这也就不难理解开启监控带来的额外巨大的开销。

Shards[23] 和 AET[8] 他们采用的采样方式还略有不同，Shards 的概念非常简单，对于一个访问的位置 L，决定这个访问位置是否被截获取决于 $hash(L)$ 是否满足下面条件：$hash(L)\ mod\ P < Q$，这样我们估算采样率大致为 $\frac{P}{Q}$。AET 的采样方式是对于完整访存序列里的每一个访问，基于概率随机选择是否监控这次访问，监控这次访问即把这次访问的地址和访问时间记录在 $hash$ 表里，等到下次访问到同样页面的时候就能够截获到一次重用，根据 $hash$ 表得到这次重用的重用时间。AET 方法不同于 Shards 的地方在于在 AET 里每一个访问都有机会被截获到，根据两种采样各自的特点，我们称 AET 采样方法为随机采样，Shards 采样方法为集合采样。

在虚拟化环境中，随机采样是很难实现的，由于获得全部的访存序列会带来巨大开销，而无法获得全部的访存序列就没有办法随机地选择页面进行跟踪，我们现在收集了所有的 PTE，就能够在特定时刻对于全部 PTE 采样置位某一部分 PTE，采样的方法我们参考了 Shards，对 PTE 地址计算 hash，符合一定 hash 特征的 PTE 被我们置位。为了克服 Shards 只会采集固定集合页面作为跟踪单元的缺点，我们利用 PTE 集合每隔一定时间周期重新扫描和随机选取，这样能够保证采样的页面不是固定的，增加了随机因素。这也是我们为什么前文提到要收集所有 PTE 的原因。

表 4.2　固定热页集采样率开销

benchmark	overhead
milc	11.3%
cactus	1.3%
soplex	3.1%
gems	7.1%
mcf	7.0%
astar	9.8%
bzip2	0.6%
bwaves	22.6%

4.3.3　动态采样率

表4.2使用的是固定热页集大小 (64 页) 以及固定采样率 ($\frac{1}{128}$) 时的开销，热页集和采样率的使用已经显著降低了客户机运行开销，但是通过表格我们发现有的测试程序的开销已经符合预期 (如 cactus,soplex,bzip2)，但是有的程序的开销还是偏大 (如 gems,milc,mcf,astar,bwaves)，我们总结当前系统仍然存在的三个问题：

- 热页集是基于程序局部性理论的，不同的程序有着不同的局部性程度，即使是在同一个程序的不同阶段，局部性也会变化。固定大小的热页集和采样率也许还不能将开销降低到我们能够接受的范围。
- 同时固定热页集大小和采样率还会带来一个现象，即比如当前程序的页面总数为 M，采样率为 $\frac{P}{Q}$，热页集大小为 H，当 $M * \frac{P}{Q} < H$ 时，截获到的页面数小于热页集大小，所有的页面都在热页集里，这时候不能截获不到任何的内存访问。这个问题本质是因为热页集大小被采样率放大，放大后的热页集大小可能会超过 100MB，这个误差是我们不允许容忍的。这个时候有两种途径改善：一是增大采样率，二是减小热页集大小。
- AET 算法是基于概率积分的算法，如果样本太小也会影响精确度，在降低开销的同时我们也要保证算法的正确性，实现中会监测一个计算周期内 AET 采集的样本数，当样本过少时我们通过采取措施增加下一个周期内的样本数量。

降低开销的途径有两个，增大热页集大小和降低采样率都能降低开销，如何在这两者中做出决定并没有理论基础和前人相关工作。很明显这是一个双变量问题，通过控制变量法，我们保持一个变量不动改变另一个变量的值来观察性能变化，表4.3表明

表 4.3　gems 不同热页集采样率下的开销

hss \ tr	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{512}$
64	15.8%	7.2%	4.1%	1.1%
128	15.5%	7.7%	3.9%	2.3%
256	15.0%	8.4%	3.6%	1.6%

表 4.4　gems 不同热页集采样率下的缺页中断次数

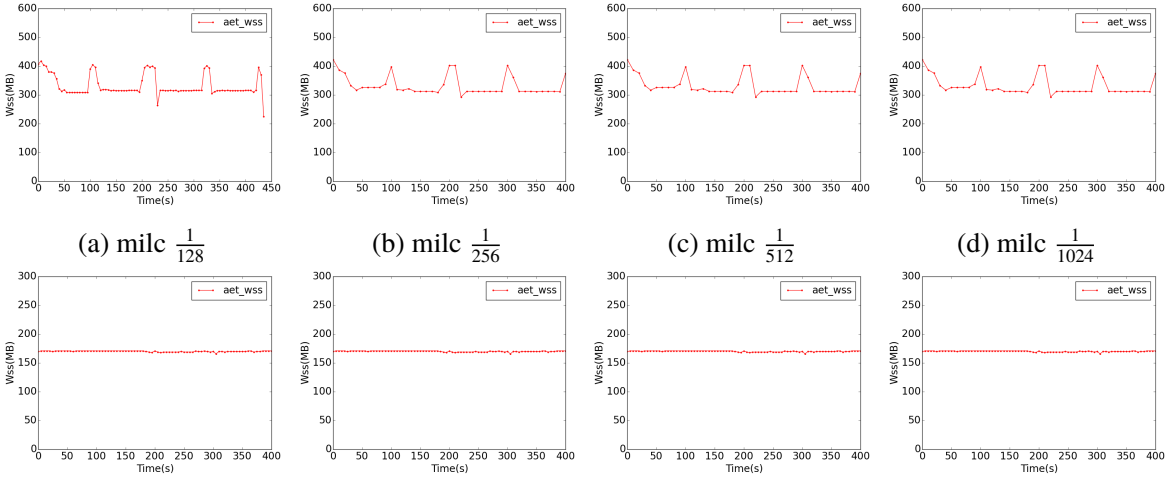
hss \ tr	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{512}$
64	26549143	13324079	6621763	2956525
128	26517108	12681288	6345955	2497447
256	25937891	12348019	5161689	1545066

对于 gems 来说降低采样率马上就降低了开销，增大热页集对于开销的改变不明显。根据我们之前的分析，程序性能的高低很有可能取决于缺页中断的次数。我们再做了一组实验，控制采样率，单纯增大热页集来分析为什么 GemsFDTD 这个程序没有性能上的提升，这次我们统计了程序运行时总的缺页中断数。

表4.4表明在增大热页集之后没有降低缺页中断次数。理论上来说增大热页集之后从热页集中淘汰出来的页面数量会减小，因为有更多的热页没有被我们系统监控，但是有下面这种情况：一个程序的热页集比我们设置的热页集要大很多，举个例子，比如如下的访问序列：“abcdefgabcdefg”，显然这段访问序列的热页大小为 7，如果当前热页集大小为 2，即使把当前热页集大小扩大一倍仍然没有降低开销，所有页面在下一次访问之前都变成了冷页，因此监控到的内存访问数并没有因为热页集大小的增加而降低。所以当一个程序他的真实热页比我们设置的热页集大小大很多或者小很多，增大热页集或者减小热页集的作用并不明显。但是持续地增大和减小最终一定会起到效果。所以调整热页集来降低开销是可以做到的，不过会存在一个慢启动的过程。

然而降低采样率直接就降低了跟踪的页面集合的大小，显然这种方式的作用效果更加直接。现在我们的系统已经能够在有限开销内实现虚拟机内存工作集预测，要真正将该系统在运行时使用还有两个问题需要解决。

- 判断我们的工作集预测系统有没有降低客户机的性能，这是系统的前提
- 如果发现客户机性能显著下降，怎么采取措施降低我们系统的开销。



(a) milc $\frac{1}{128}$　(b) milc $\frac{1}{256}$　(c) milc $\frac{1}{512}$　(d) milc $\frac{1}{1024}$

(e) sjeng $\frac{1}{128}$　(f) sjeng $\frac{1}{256}$　(g) sjeng $\frac{1}{512}$　(h) sjeng $\frac{1}{1024}$

图 4.4　不同采样率下的工作集变化曲线

由上面的实验和分析，我们系统采用动态采样率来降低开销，在介绍动态采样率实现之前，我们需要验证不同采样率下对精度的影响。我们的实验采用的是固定热页集大小 (64page)，在不同采样率 ($\frac{1}{128}$，$\frac{1}{256}$，$\frac{1}{512}$，$\frac{1}{1024}$) 下 SPEC CPU 2006 部分测试程序的工作集变化曲线如图5.6。这个工作集变化曲线是通过每一个时刻的 MRC 曲线，然后计算 5% 失效率对应的页面数作为工作集大小而绘制的在整个 benchmark 运行期间的工作集变化情况。我们实验表明在各种采样率下工作集的变化曲线基本保持一致，误差很小。AET[8] 指出在万分之一采样率仍然能够保持较高精度。有了这些实验数据和理论认证，接下来的工作便是寻找一个合适地动态采样率算法。

动态采样率第一件事情是要先确定一个初始的采样率大小。过大或者过小的采样率都会带来不好的结果。太大的采样率会使得在初始时候开销很大，并且调整采样率是需要花费时间的，如果调整采样率的时间间隔为 5 秒，即采样周期为 5 秒，需要 6 次调整才能够将开销降低到足够小，那么就会浪费 30 秒的时间在调整采样率上。最严重的还是在初始采样率下由于起初采样率一般会设置得较高，客户机的性能得不到保障。而太小的采样率一是影响精度，二就是会出现前文提到的热页集放大导致热页集里的页无法从热页集剔除，进而截获不到任何访问。通过在我们环境中进行试验测试，我们选择 $\frac{1}{128}$ 作为起始的采样率，热页集大小固定为 64 个页，我们的客户机内存大小配置为 4GB。对 SPEC CPU 2006 所有 benchmark 以始终 $\frac{1}{128}$ 的采样率进行工作，得到的平均开销为 6.8%，不过这是在客户机内存大小为 4GB 下面的配置，如果虚拟机配置的内存更高，建议适当降低初始的采样率。

根据[29] 的实验结果，GemsFDTD，Milc 和 Mcf 有着很高的开销，所以我们把这几个 benchmark 作为实验的一部分，另外我们再从 SPEC CPU 2006 随机选取了五个

表 4.5　热页集 64 页采样率 $\frac{1}{128}$ 下的开销

测试程序	标准时间 (s)	运行时间 (s)	访存数	缺页中断数	缺页中断比例	开销
gems	394	455	9.223E+11	26517108	1.446E-05	15.5%
milc	387	453	4.177E+11	30116393	3.499E-05	17.0%
mcf	310	353	1.460E+11	17879310	5.803E-05	13.8%
cactus	593	601	1.950E+12	975016	4.999E-07	1.3%
soplex	199	205	2.615E+11	1383318	5.28E-06	3.1%
lbm	217	224	3.491E+11	2441754	6.993E-06	3.3%
sjeng	385	397	7.759E+11	3961062	5.104E-06	3.1%
omnetpp	267	295	2.589E+11	13369127	5.162E-05	10.6%
fake	194	197	8.585E+11	1352064	1.574E-06	1.5%

benchmark：Cactus，Soplex，Lbm，Sjeng 和 Omnetpp。另外我们自己写了一个伪测试程序。我们还是固定了采样率 $\frac{1}{128}$，热页集 64 页来进行实验。

从表4.5能够看出 GemsFDTD，Milc，Mcf 和 Omnetpp 开销偏大，另外五个测试程序的性能还是比较出色的，我们期望我们的内存预测系统的开销控制在 3% 以内。更高的开销就无法接受了，表4.5也同样收集了内存访问数和缺页中断次数，我们发现当缺页中断占内存访问数的比例小于 $1 * 10^{-6}$ 时展现出不错的性能。这个指标也许能够帮助我们估计客户机的性能损失，指导动态采样率的过程。这个值也是具有实际意义的，它表示每多少个内存访问会发生一次缺页中断，这对于操作系统来说也是一个优化性能的重要参数。我们把这个比值作为衡量系统对于客户机性能影响的指标，也同时作为调整采样率的指导参数。

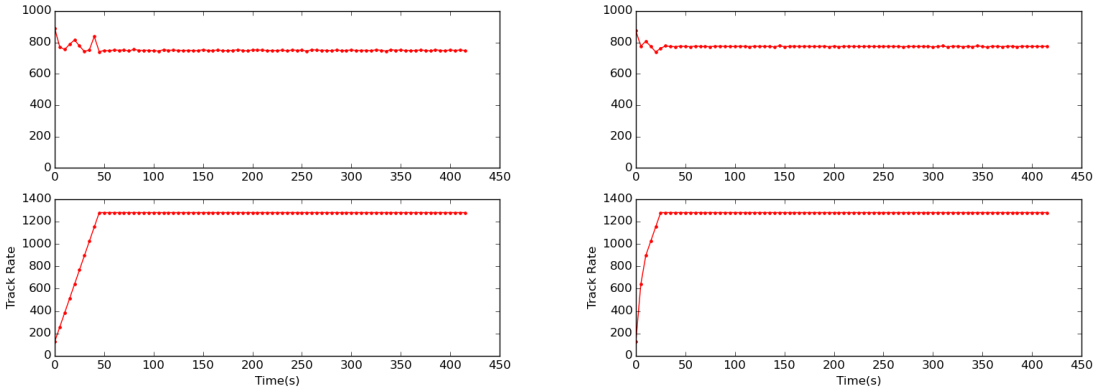
我们实验发现 $1 * 10^{-6}$ 是一个不错的性能分界线，我们把这个值称为 Sr。如果在上一个采样时间段里，缺页中断比例比 Sr 大，则需要降低采样率，我们定义 $\frac{1}{T}$ 为采样率以及 r 作为实际运行过程中的缺页中断和内存访问数的比例，那么我们可以得到一个相当简单的调整算法：

$$T = T + 128 \qquad if \quad r > Sr \tag{4.2}$$

这个算法对于大的工作集比如 GemsFDTD 来说工作得不错，在监控之初，缺页中断的比例高于 Sr，采样率也从 $\frac{1}{128}$ 逐渐下降到了 $\frac{1}{1280}$，开销也从 7.2% 下降到了 2.3%，但是仍然使用了 10 个周期才将采样率调整合适。调整如果太缓慢会影响到测试程序初期的性能，一个直观的想法是如果在一个采样周期得到的缺页中断比例和理想值差距

越大，采样率下降的速度就越快。我们使用了下面的指数降低采样率的算法4.3，同样对于 GemsFDTD，整体的开销进一步降低到 0.84%。

$$T = T + 128 * (\log(\frac{r}{Sr}) + 1) \qquad if \quad r > Sr \tag{4.3}$$



(a) gems 采样率线性下降　　(b) gems 采样率指数下降

图 4.5　两种算法运行示例

图4.5中左图采样率线性下降，右图中采样率指数下降，能够看出指数算法采样率收敛到合适的值会更快，所以它整体的开销会更低。

最后一个问题是当采样率降低到一定程度之后，如果当前工作集突然变得很小时会出现什么情况？以 GemsFDTD 为例，当采样率降低到 $\frac{1}{1280}$，此时由于它使用的总的页面数很大，在这种采样率之下能够监控 167 个页。但是假如 GemsFDTD 释放掉它使用的绝大多数内存，只是用了 100MB 内存即 25600 个页，那么在当前这个采样率下只能监控到 20 个页面。由于热页集大小固定为 64 个页，所以监控的页面都在热页集里面，也就是说我们无法截获任何的页面访问，更不会得到任何的页面重用。

图4.6是 fake benchmark 的工作集曲线图，fake benchmark 原本有七个阶段，在经过动态采样率调整后，采样率很快降低到了 $\frac{1}{512}$，fake benchmark 最后一个阶段将会扫描 100MB 页面，系统无法预测出这个阶段的内存工作集。这是因为 100MB 内存对应 25600 个页，在采样率为 $\frac{1}{512}$ 的时候监控 50 个页，然而热页集大小为 64 个页，所以我们监控的所有页面都在热页集中，因此截获不到任何的页面访问，在工作集变化曲线中体现为最后一个阶段监控不到内存工作集的大小。

解决办法是如果在一个采样周期之后我们发现我们截获到的页面访问数偏小，那么很有可能客户机的内存使用量相比于之前有了减小，应当考虑适度增大采样率。增大采样率并不像降低采样率那样紧急，因为如果截获不到任何页面访问，那么也就不会给客户机带来额外的性能影响，并且通过热页集和采样率我们能够估算出客户机内

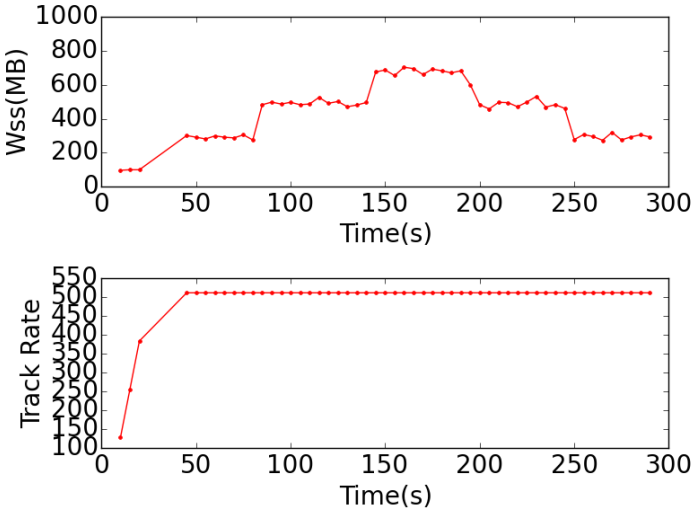


图 4.6　动态采样率下的 fake benchmark

存使用量的上界。并且由于 AET 算法是一个基于概率的模型，我们定义所能够容忍的一个采样周期里最少的页面截获数为 P，当实际截获到的页面数 n 比 P 要小的时候，我们使用算法4.4的办法增大采样率。

$$T = T - 128 \qquad if \;\; n < P \tag{4.4}$$

综合上面所述的两种调整采样率情况，我们得到了最后的动态调整采样率算法如下4.5

$$T = \begin{cases} T + 128 * (\log(\frac{r}{Sr}) + 1) & r > Sr \\ T - 64 & n < P \end{cases} \tag{4.5}$$

第五章　实验

本章的内容主要是围绕我们内存工作集预测系统的准确性验证和开销分析展开，通过大量实验验证系统的正确性以及在加入各种优化之后的稳定性，验证系统能够通过优化算法使得能够在低开销的情况下获得较为精准的预测结果。

5.1　实验环境

我们实验用到的机器配置是 Intel Core I7 处理器，16GB 的内存，四个支持超线程的 2.8GHz 核心。我们所有的实验都是在 Xen 虚拟化环境下进行的，Xen 的版本使用的是 4.5.1，Linux Kernel 的版本是 4.2.1，为了排除多个虚拟机对于处理器的竞争，每一个虚拟机都被绑定到一个固定的核上。每个虚拟机配置的内存大小为 4GB。我们实验选择的监控周期为 5 秒，每隔 5 秒钟，根据得到重用信息绘制得到 MRC 曲线。

我们的跟踪系统是一个通用的实现了热页集和采样机制的跟踪系统，有了这样一个跟踪系统，运行什么算法去获得 MRC 都是容易实现的。LRU 算法的话由于需要获得重用距离，就要人为模拟 LRU 替换过程，每次重用时获得它在 LRU 里的位置；AET 算法需要获得重用时间，即在本次访问的时候将当前的访问时间记录在 hash 表里，等到重用时从 hash 表中获取上次访问时间得到重用时间并更新 hash 表信息。为了比较在真实系统下 AET 算法的准确性，我们需要比较这两个算法所计算出的 MRC。为了保证在同样访存序列下比较 AET 算法是否能够达到 LRU 的精度，我们在截获一个页面的时候同时通过两种算法得到重用时间和重用距离，完成两个算法相关的操作。

Algorithm 1 Fake benchmark

```
malloc N MB memory organized as an array;
define a list L that indicates memory usage in each step;
for each i ∈ L do
    scan the first i MB array of N several times;
end for
```

我们使用了两种测试程序，一种测试程序是我们手动写的，如算法1所示，我们称之为伪测试程序。这个伪测试程序相当简单，起初分配一段固定大小的内存，顺序以及周期性地扫描我们分配的内存区域。通过分配不同的内存大小能够制造更多的测试程序。为了增加伪测试程序工作集大小变化，我们将伪测试程序分为了多个阶段，在不同的阶段扫描的内存范围不同，表5.1显示了不同程序阶段扫描的内存大小。另一类测试程序出自 SPEC CPU 2006，这些测试程序更加地通用以及符合实际。

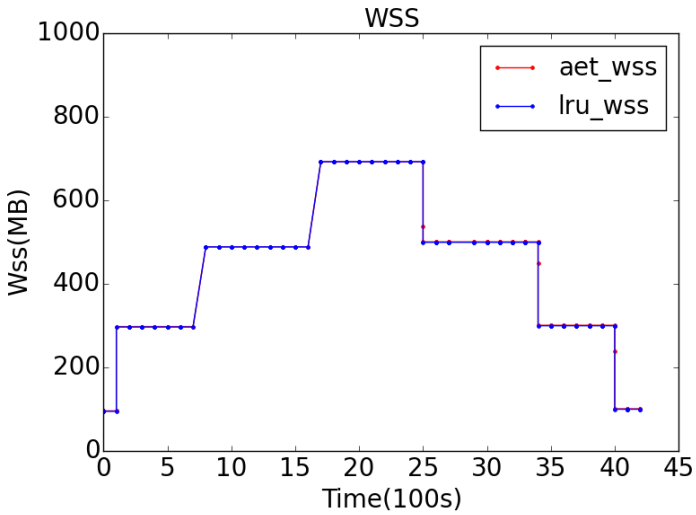


图 5.1　fake benchmark

表 5.1　fake benchmark 各阶段内存使用

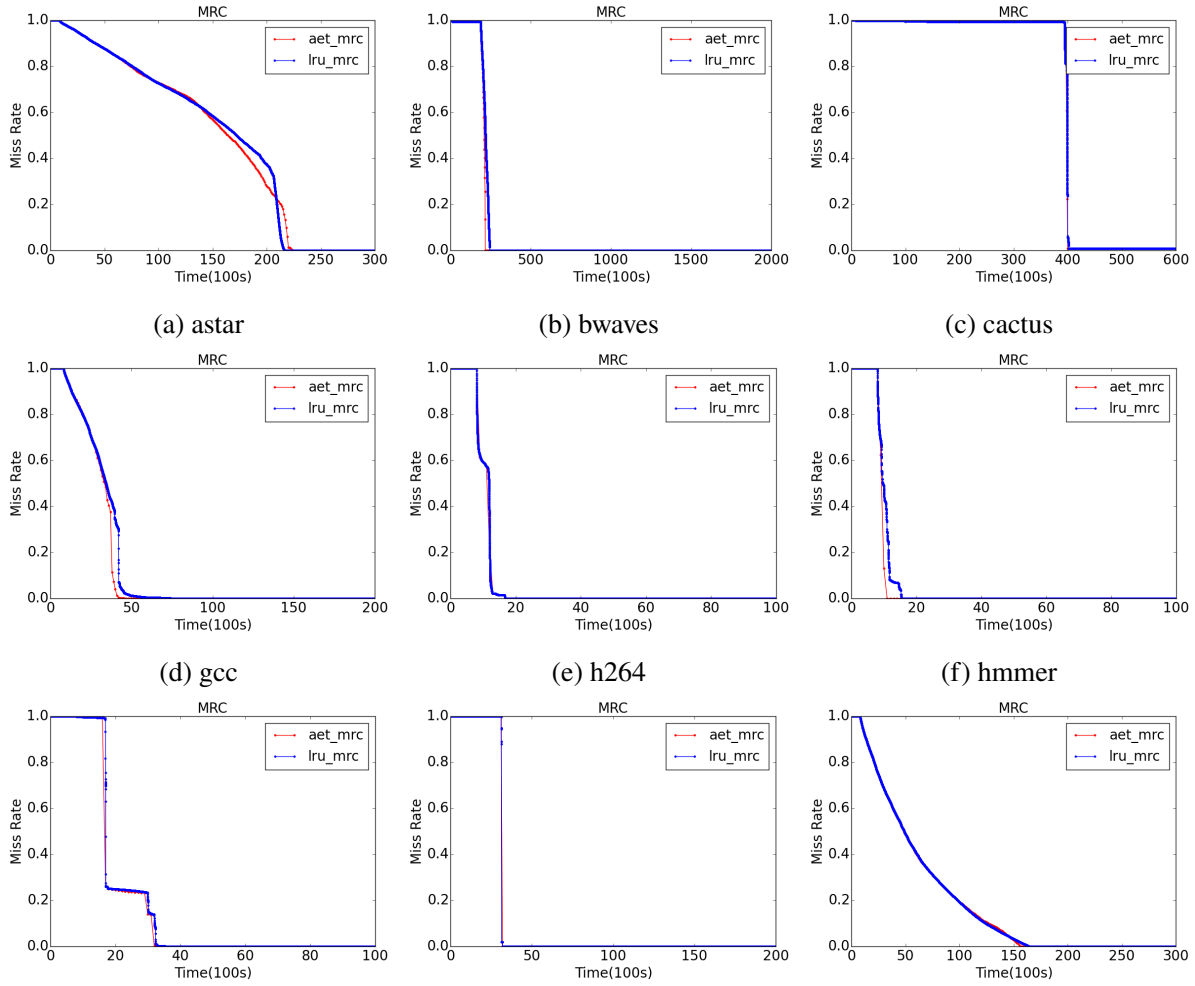
step 1	step2	step3	step4	step5	step6	step7
100MB	300MB	500MB	700MB	500MB	300MB	100MB

5.2　正确性验证

5.2.1　和 LRU 的比较

Hu[8] 使用“master MSR”序列[26] 验证了 AET 的正确性，使用 $1 * 10^{-6}$ 作为采样率的 AET 算法计算出的 MRC 曲线和用 LRU 计算出的 MRC 曲线的平均绝对错误 (Mean Absolute Error，MAE) 只有 0.01，不过这个“master MSR”序列是由 Microsoft Research Cambridge[14] 提供的存储访问序列，并不是内存访问序列。MEB[25] 使用优化的 LRU 算法和热页集得到精确的 MRC 曲线。把前面两者的工作结合起来，要验证 AET 模型在内存工作集预测的可行性，就要在同样的内存访问序列下比较两种算法计算的 MRC 曲线的差别。要实现这个比较并不难，当我们系统截获到一个我们人为制造的缺页中断时，同时维护两套数据结构，一套用作 LRU 的计算，一套用作 AET 的计算，每一个周期之后利用 LRU 或者是 AET 收集的数据分别计算 MRC。我们的实验参数采用的 MEB[25] 的配置，热页集大小设置为 2048 页，MEB 并没有采用采样技术，所以在这组正确性验证实验中，我们并没有开启采样。

图5.1清晰地反映出伪测试程序有七个运行阶段，虽然 AET 模型使用的是重用时间分布，LRU 算法使用的使用距离分布，但是在我们这个伪测试程序中，内存访问是



(a) astar　(b) bwaves　(c) cactus

(d) gcc　(e) h264　(f) hmmer

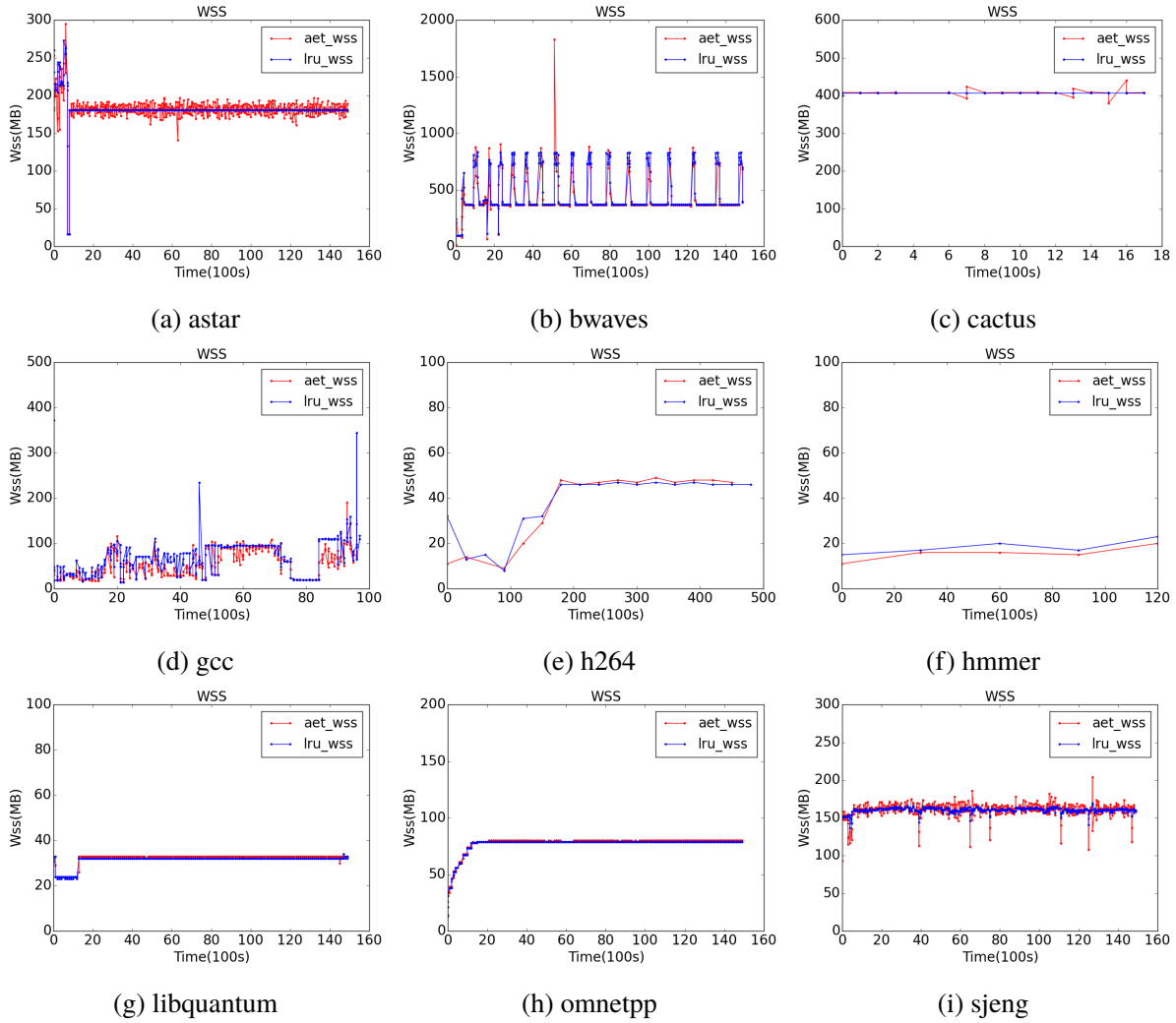
(g) libquantum　(h) omnetpp　(i) sjeng

图 5.2　运行时刻 MRC

顺序的且是循环的。因此 LRU 算法里的重用距离就是整个内存的大小，而在 AET 模型中，我们用逻辑时间作为时间度量，重用时间即为两次对同一地址的访问之间间隔的内存访问数，所以重用时间也是整个内存的大小。伪测试程序得到的重用时间分布和重用距离分布具有同样的分布，在这种情况下，对于工作集的预测是完全准确的。

我们还要验证一下使用真实的工作负载下 AET 模型正确性，为了排除采样率的影响，同样我们没有开启采样，热页集的大小设置为 2048。因为我们的监控周期是 5 秒钟，在整个程序运行的过程中会计算出很多的 MRC 曲线，我们从中随机选取一些 MRC 对比曲线如图5.2所示。

和伪测试程序不同的是，在真实的工作负载下，页面的重用距离往往是小于重用时间的，这是因为在相同地址两次相邻访问之间，可能会存在别的内存访问发生了重用，这就导致该地址在 LRU 里的深度小于重用间隔。两种算法最终计算得到的 MRC



(a) astar (b) bwaves (c) cactus

(d) gcc (e) h264 (f) hmmer

(g) libquantum (h) omnetpp (i) sjeng

图 5.3 工作集曲线图

曲线具有很高的相似性。利用 MRC 曲线，我们定义工作集为失效率为 5% 下的内存大小，那么每一个监控周期之后我们都能得到一个预测的工作集大小，将这些工作集大小和时间对应就能够绘制出工作集变化走势图，我们就能够得到测试程序在整个执行过程中的内存变化情况，这对于动态调整内存大小是有指导意义的。所以我们也比较了两种算法下的工作集变化走势图。

如图5.3所示，AET 算法是基于概率的模型，通过 AET 算法计算的 MRC 曲线和用 LRU 算法计算的 MRC 曲线会有一点偏差，这就导致在取 5% 失效率为工作集大小的时候，两种算法对于工作集大小的预测也会有所偏差，不过两种算法得到的工作集变化曲线整体趋势和大小相近，误差在可接受的范围内。

在这组实验中，我们发现同时计算 LRU 和 AET 的测试程序的运行时间是原本运行时间的数倍，由于我们只是希望能够对比 AET 算法的正确性，所以我们 LRU 算法

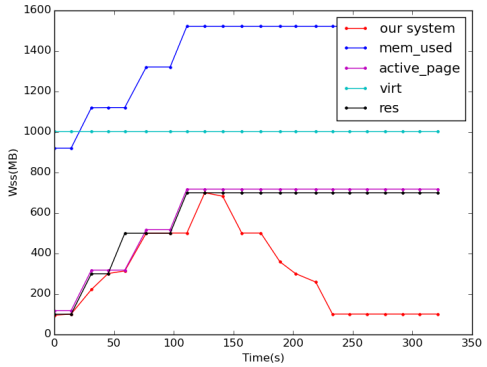
的实现采样的最朴素的双向链表的实现，每一次内存访问都需要在 LRU 双向链表里查找本次访问地址的 LRU 深度，时间复杂度为 O(N)(N 为总的不同访问地址总和)，而没有使用类似线段树、AVL 平衡树等算法 (时间复杂度为 $O(log_N)$)，而 AET 算法每次更新时间重用距离分布的时间复杂度为 $O(1)$。为了充分验证正确性而又能够在一定时间得到结果，我们只截取了一个测试程序前 1000 个采样周期里的结果。我们还发现一个有趣的事实，虽然监控周期都是 5 秒，但是在这组对比实验中，5 秒测试程序执行的指令数是远远不及没有加入 LRU 算法时候测试程序 5 秒执行的指令数的，这就好比开启了慢放功能，所以这组实验得到的 MRC 曲线和没有使用 LRU 算法作为对比得到的 MRC 曲线是没有可比性的，因为他们的时间维度不同，通常来说，慢放之后 5 秒钟里面访问的内存数会减小，所以预测得到的工作集大小会小于正常情况下测试程序 5 秒钟的工作集大小。

5.2.2　和真实工作集比较

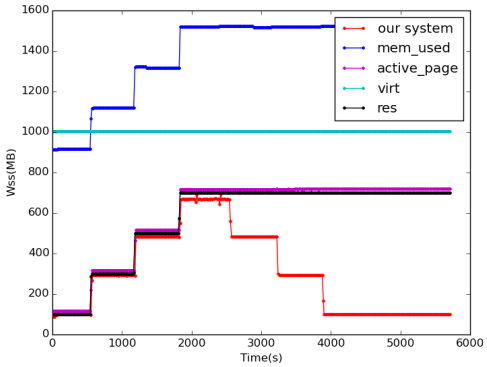
因为 MRC 曲线的经典算法就是 LRU 算法，我们将实现在我们系统里的 AET 算法和 LRU 算法得到的 MRC 曲线进行比较，就能够验证 AET 模型在我们内存工作集预测上的可行性。但是上一节我们也提到过在我们引入 LRU 算法计算后大大增加了系统的开销，导致测试程序运行的时间大大增加，最后得到的关于测试程序的工作集变化曲线和真实的曲线会有偏差。由于已经验证了 AET 算法本身在系统中的计算正确性，所以为了和真实的程序工作集比较，我们系统关闭 LRU 算法仅仅使用 AET 算法，同样热页集大小还是固定为 2048 个页，没有开启采样。

对于真实工作集的获得，[25][29] 是两篇关于内存工作集预测的文章，但是这两篇文章并没有明确地指出 SPEC CPU 2006 各个测试程序运行时候的工作集变化情况，[29] 通过它们自己制造的两个随机测试程序 random 和 mono 来验证正确性。由于之前并没有太多的关于内存工作集预测的研究，所以我们也是通过本章一开始提到的伪测试程序来验证系统的可靠性，对于 SPEC CPU 2006 里的测试程序，我们通过和系统所给出的内存使用量来大致进行比较。

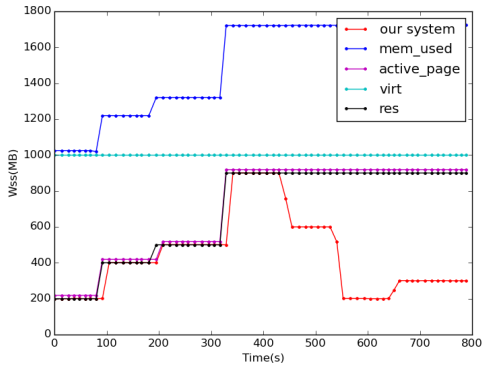
图5.4的四幅图中上面一排两个图是固定各个阶段扫描的内存大小，下面一排两个图是随机地生成各个阶段扫描的内存大小，两次随机扫描的各个阶段的内存大小分别为 [200,400,500,900,600,200,300]MB，[800,300,300,600,200,700,600]MB，左边的图在扫描的内存范围里采用顺序扫描，右边的图则是随机扫描，之所以要增加顺序扫描和随机扫描这一组对比实验，是因为顺序扫描各个阶段的重用时间都是固定的，而随机扫描各个阶段的重用时间不是固定的，我们从某一时刻得到的重用时间分布直方图中也能够看出来。由于我们在 fake 程序初 malloc 的大小为 1G，所以从程序一开始 virt 显示



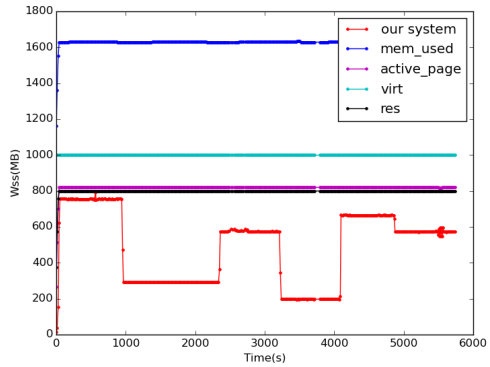
(a) Fixed Stage Sequential Scan



(b) Fixed Stage Random Scan



(c) Random Stage Sequential Scan

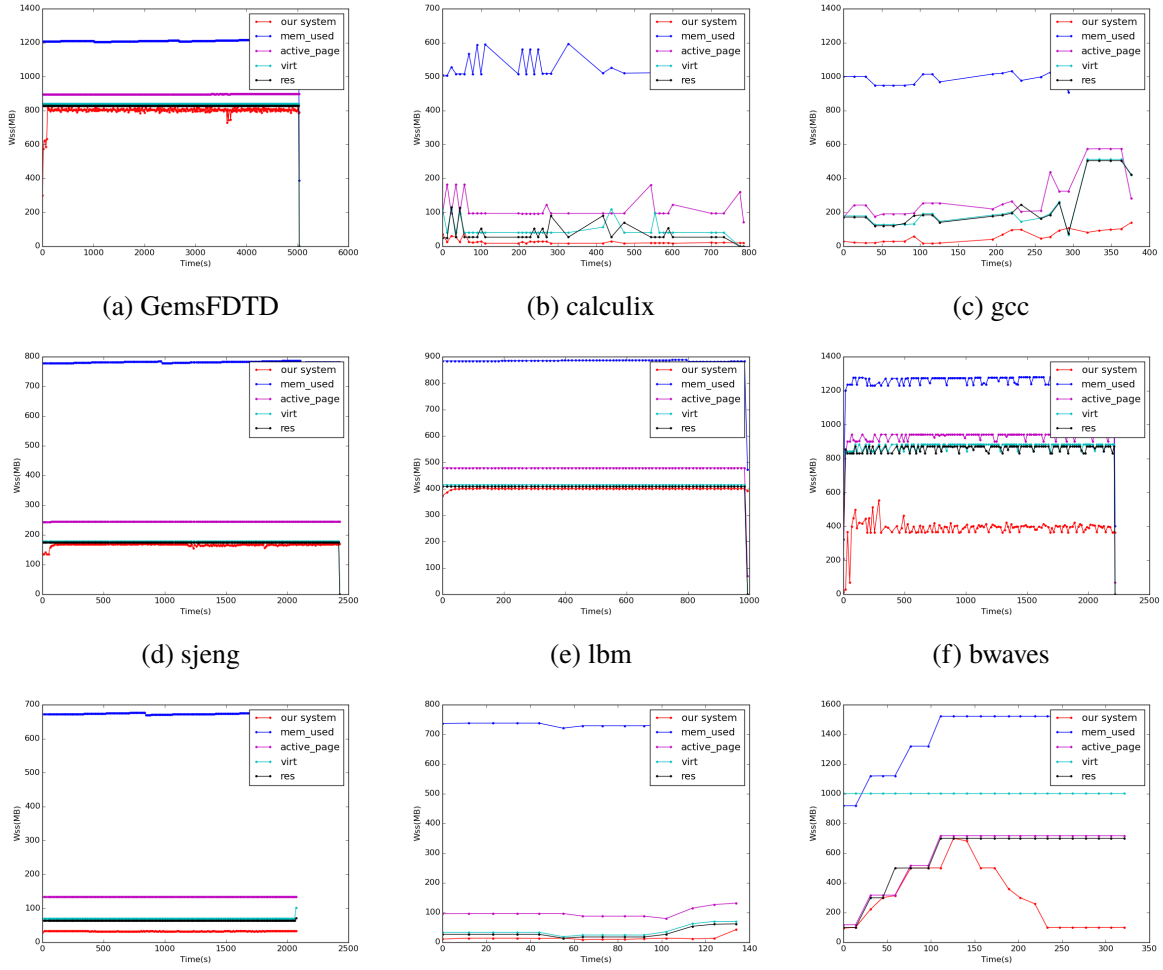


(d) Random Stage Random Scan

图 5.4　Fake benchmark 预测工作集曲线图和系统参数比较

的值就是 1G，并且无论我们使用内存的大小怎么变化，virt 的值也都不会改变。res 和 free memory 这两个线条随着阶段性页面使用的增加而增大，res 和使用的页面数相当，这也反映了操作系统请求调页的机制，当真正使用内存的时候才会申请页面。但是当第四个阶段过去之后，res 值就保持不变了，因为这部分申请的内存虽然没有使用，但是也没有被替换到 SWAP 空间去，所以 res 仍然会统计到这部分实际上并没有使用的页面。

而对于真实测试程序来说，我们不易得到工作集大小，因为操作系统并不会帮助我们计算也没有相应的硬件帮助，操作系统会记录页面分配情况，由于有页面替换，所以还有活跃页面 (active) 和不活跃页面 (inactive) 的统计。如图5.5所示，我们使用的系统参数有内存总数，剩余的内存数，通过做差得到了使用的内存数，活跃页面数，这些是整个系统相关的内存参数，还用到了进程级别的系统监控参数，virt 和 res，virt 是进程申请的虚拟内存空间的大小，它包括所有的代码，数据和共享库，加上已换出的页面，res 是一个任务正在使用的没有交换的物理内存也通常称为驻留内存。整个系统里关于内存的参数通常会比实际使用到的内存要大，因为整个系统还包括文件缓存，所



(a) GemsFDTD　(b) calculix　(c) gcc

(d) sjeng　(e) lbm　(f) bwaves

(g) libquantum　(h) h264　(i) fake

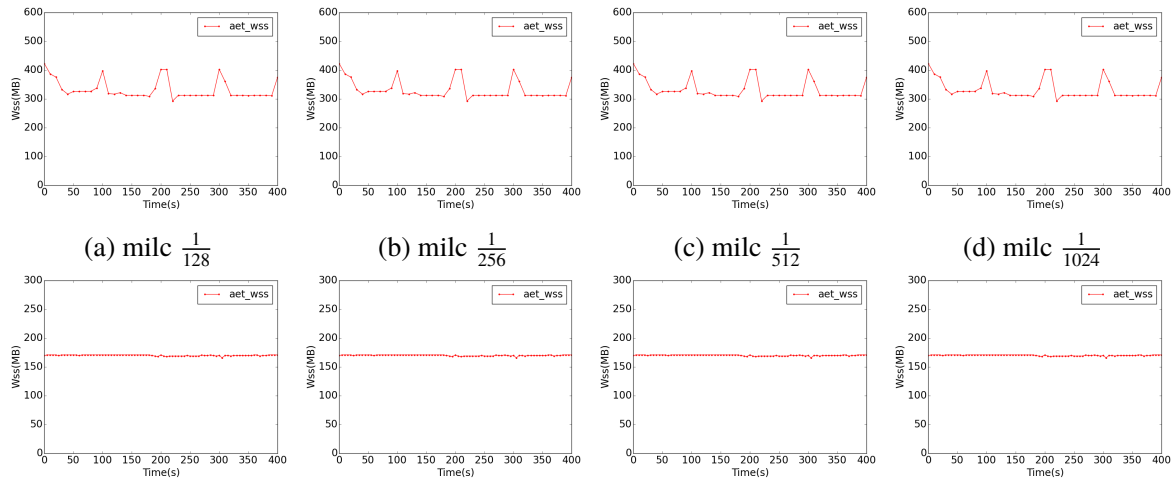
图 5.5　预测工作集曲线图和系统参数比较

有后台进程用到的内存，而当前正在运行的测试程序显然不会使用到这么多的内存数。virt 和 res 是进程级别的统计，我们读取出正在运行的测试程序的相关数据，virt 是该测试程序申请的虚拟空间大小，res 是它实际使用到的页面数，但是这两个数并不能反映当前工作集，他们是整个测试程序从载入到运行到此刻的所申请的内存总和，而往往当前的工作集里并不会使用到所有申请到的内存。我们系统计算的是当前负载的工作集，所以我们的计算结果通常会比系统参数要小。通过和系统参数比较，我们也能够验证我们系统得到的结果的正确性。

5.2.3　不同采样率下的准确性

在全虚拟化环境下要实现内存工作集预测的开销往往是不可接受的，我们系统的目标也就是为了降低开销到可接受的范围，我们后文开销分析里面提到了在不开启采

样下平均开销达到 557.75%，这显然是不可接受的。上面的实验已经验证了无采样情况 AET 的准确性，下面就需要验证一下在各种不同采样率下系统得到的工作集是否符合预期。由于开启采样与否以及不同采样率之间程序执行的快慢是不一的，所以我们尽量挑选一些在上一节实验中得到的工作集变化曲线不那么明显的测试程序进行测试。关于热页集的选择我们这里没有继续使用 2048，对于我们的动态采样率算法来说，我们都是固定热页集大小为 64 个页，所以为了验证动态采样率下的准确性，我们也把这组实验的热页集大小设置为 64 个页。



(a) milc $\frac{1}{128}$ (b) milc $\frac{1}{256}$ (c) milc $\frac{1}{512}$ (d) milc $\frac{1}{1024}$

(e) sjeng $\frac{1}{128}$ (f) sjeng $\frac{1}{256}$ (g) sjeng $\frac{1}{512}$ (h) sjeng $\frac{1}{1024}$

图 5.6　不同采样率下的工作集变化曲线

从图5.6中我们看到如同预期采样率越低，运行的时间越短，并且各种采样率下的工作集变化曲线偏差不大，由于测试程序各个监控周期里的工作集大小基本不变，我们计算了各种采样率下的各个监控时段的工作集均值，和没有开启采样下的工作集均值，误差为 xx，在降低这么多开销的情况下，这点误差是可以接受的。我们在这个实验中设置的采样率最大为 $\frac{1}{128}$，最小为 $\frac{1}{1280}$，这也是我们动态采样率算法所使用到的采样率的上界和下界。

5.3　开销分析

内存工作集预测最大的难点就是开销，所以这也是我们系统极力想要克服的难点。由于缺乏对操作系统内存访问的感知，我们只能通过人为制造缺页中断的办法来截获系统的内存访问，在截获内存访问之后，完成一些特定的算法，最后完成对内存工作集大小的预测。所以时间开销是由于人为制造的缺页中断而引起的相关计算所导致，假设我们一共截获了 N 个人为的缺页中断，那么缺页中断本身所需要的上下文切换和中

断处理开销为 M，完成算法计算的开销为 T，那么总的时间复杂度就是 N*(M+T)，M 的时间是常数，对于 T 的时间开销计算，如果采用 LRU 算法，每次算法执行的开销为 $O(P)$(P 为不同的访存地址数)，采用优化的平衡树算法的开销为 $O(log_P)$，而 AET 算法的开销则是常数，所以理论上我们系统的开销应该和认为截获的缺页中断数相关，我们的实验通过固定热页集为 64 个页，不同采样率来控制缺页中断次数，我们统计了缺页中断数的比例以及开销。

受限于篇幅，表格5.2只列出了五个 benchmark 在不同采样率下的数据，我们统计的 page fault 的比例是 page fault 占内存访问数的比例，不难发现，人为缺页中断次数基本上和采样率的大小成正比的关系，当 page fault 的比例小于 10^{-6} 时开销基本上可以忽略不计了。所以我们才有了动态调整采样率这个想法，我们将动态采样率算法应用在所有 SPEC 2006 测试程序上得到的平均开销为 1.4%，各 benchmark 详细开销见表格5.3。

表 5.2　不同采样率下的开销

测试程序			
gems	采样率	缺页中断比例	开销
	$\frac{1}{128}$	1.446E-05	7.1%
	$\frac{1}{256}$	7.195E-06	4.1%
	$\frac{1}{512}$	3.214E-06	1.1%
	$\frac{1}{1024}$	1.981E-06	3.6%
milc	采样率	缺页中断比例	开销
	$\frac{1}{128}$	3.499E-05	11.3%
	$\frac{1}{256}$	1.801E-05	7.7%
	$\frac{1}{512}$	109713.087	5.6%
	$\frac{1}{1024}$	3.978E-06	3.3%
mcf	采样率	缺页中断比例	开销
	$\frac{1}{128}$	5.803E-05	7.0%
	$\frac{1}{256}$	2.432E-05	1.9%
	$\frac{1}{512}$	4.561E-06	1.6%
	$\frac{1}{1024}$	8.626E-06	1.2%
soplex	采样率	缺页中断比例	开销
	$\frac{1}{128}$	5.288E-06	3.1%
	$\frac{1}{256}$	2.260E-06	2.1%
	$\frac{1}{512}$	9.300E-07	1.1%
	$\frac{1}{1024}$	1.746E-06	0.1%
omnetpp	采样率	缺页中断比例	开销
	$\frac{1}{128}$	5.162E-05	10.6%
	$\frac{1}{256}$	1.688E-05	4.0%
	$\frac{1}{512}$	2.244E-07	1.3%
	$\frac{1}{1024}$	7.046E-07	0.2%

表 5.3　动态采样率下的开销

测试程序	标准时间 (s)	运行时间 (s)	缺页中断比例	开销
459.gems	393.7	411.0	7.23 E-06	4.1%
433.milc	387.0	402.0	7.09 E-06	4.4%
429.mcf	310.0	317.0	7.24 E-06	0.1%
436.cactusADM	593.0	603.0	4.99 E-07	1.7%
450.soplex	198.7	205.0	5.60 E-06	2.2%
473.astar	310.3	320	4.21813E-06	3.1%
401.bzip2	345.0	342.0	4.17 E-06	0.1%
410.bwaves	444.0	452.0	3.37 E-06	1.6%
403.gcc	231.0	238.0	4.10 E-06	3.0%
434.zeusmp	350.0	353.0	2.55 E-06	0.6%
470.lbm	216.6	225.0	6.91 E-06	3.2%
454.calculix	593.0	601.0	3.78 E-07	0.8%
456.hmmer	328.0	333.0	7.61 E-07	0.3%
458.sjeng	385.7	397.0	4.28 E-06	0.6%
462.libquantum	323.3	336.0	9.69 E-06	2.7%
464.h264ref	393.0	406.0	2.59 E-06	0.5%
465.tonto	496.7	503.0	1.58 E-06	0.5%
471.omnetpp	266.6	276.0	1.15 E-05	1.8%
482.sphinx3	430.7	432.0	5.28 E-07	1.0%
416.gamess	605.0	614.0	4.47 E-08	1.2%
435.gromacs	355.0	366.0	8.11 E-07	0.3%
437.leslie3d	297.0	308.0	2.76 E-06	0.7%
444.namd	304.0	307.0	2.89 E-07	0.1%
445.gobmk	334.0	343.0	2.02 E-06	0.1%
453.povray	123.0	125.0	4.56 E-07	0.8%
Mean				1.4%

第六章　结论

本文针对虚拟化场景，实现了虚拟机的实时内存预测，这项技术能够为虚拟机提供者提供内存动态调配的依据，对于物理资源的整合和优化配置起到重要作用。

虚拟化场景下对虚拟机内存预测的难点主要在于虚拟机的运行对于底层虚拟机管理器来说是透明的，本文通过标记虚拟机页表项人为制造缺页中断使得虚拟机管理器获得虚拟机内存使用情况，不过这引来了巨大的开销，因为每次内存访问都要被虚拟机管理器截获并做相应的处理。基于 AET 算法提出的采样方法，本文通过采样只对部分页面进行跟踪，由于 AET 算法只需要重用时间分布就能够计算失效率曲线 (MRC)，而随机采样得到的重用时间分布是近似于真实重用时间分布，本文经过实验验证了在各种采样率下得到的 MRC 的一致性，从而证明了 AET 采样方法对于虚拟机内存预测的可行性。

但是固定采样率仍然带来了两个问题，采样率过高会导致虚拟机性能的损失，采样率过低我们得到的重用时间分布会失真于真实重用时间分布，为了控制开销而又能保证预测的准确性，本文提出了动态采样率算法，通过一个监控周期里的缺页中断次数和内存访问数的比例来保证虚拟机的性能，通过控制一个采样周期里单位时间缺页中断次数来保证采集到的内存访问的次数从而保证重用时间分布近似于真实重用时间分布。通过对 SPEC CPU 2006 所有 benchmark 进行测试，动态采样率对于虚拟机性能的损失平均仅为 1.4%

参考文献

[1] R. Azimi, L. Soares, M. Stumm *et al.* "*Path: page access tracking to improve memory management*". In: *International Symposium on Memory Management*, **2007**: 31–42.

[2] B. T. Bennett and V. J. Kruskal. "*LRU stack processing*". *IBM Journal of Research and Development*, **1975**, *19*(4): 353–357.

[3] D. Bruening, E. Duesterwald and S. Amarasinghe. "*Design and implementation of a dynamic optimization framework for Windows*". In: *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, **2001**.

[4] P. J. Denning. "*The working set model for program behavior*". *Communications of the Acm*, **1968**, *26*(1): 43–48.

[5] C. Ding and X. Xiang. "*A higher order theory of locality*". In: *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, **2012**: 68–69.

[6] X. Hu, X. Wang, Y. Li *et al.* "*LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache.*" In: *USENIX Annual Technical Conference*, **2015**: 57–69.

[7] X. Hu, X. Wang, Y. Li *et al.* "*Optimal Symbiosis and Fair Scheduling in Shared Cache*". *IEEE Transactions on Parallel and Distributed Systems*, **2017**, *28*(4): 1134–1148.

[8] X. Hu, X. Wang, L. Zhou *et al.* "*Kinetic Modeling of Data Eviction in Cache*". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, **2016**: 351–364. https://www.usenix.org/conference/atc16/technical-sessions/presentation/hu.

[9] Intel. *Intelő 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, **2016**.

[10] J. M. Kim, J. Choi, J. Kim *et al.* "*A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references*". In: *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, **2000**: 9.

[11] H. Luo, P. Li and C. Ding. "*Thread Data Sharing in Cache: Theory and Measurement*". In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, **2017**: 103–115.

[12] D. Magenheimer *et al.* "*Memory overcommit... without the commitment*". *Xen Summit*, **2008**: 1–3.

[13] R. L. Mattson, J. Gecsei, D. R. Slutz *et al.* "*Evaluation techniques for storage hierarchies*". *Ibm Systems Journal*, **1970**, *9*(2): 78–117.

[14] D. Narayanan, A. Donnelly and A. Rowstron. "*Write OFF-loading: Practical power management for enterprise storage*". In: *Usenix Conference on File and Storage Technologies*, **2008**: 256–267.

[15] R. H. Patterson, G. A. Gibson, E. Ginting *et al. Informed prefetching and caching*. ACM, **1995**.

[16] M. K. Qureshi and Y. N. Patt. "*Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches*". In: *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, **2006**: 423–432.

[17] S. Rajasekaran, S. Duan, W. Zhang *et al.* "*Multi-cache: Dynamic, Efficient Partitioning for Multi-tier Caches in Consolidated VM Environments*". In: *Cloud Engineering (IC2E), 2016 IEEE International Conference on*, **2016**: 182–191.

[18] I. Stefanovici, E. Thereska, G. O'Shea *et al.* "*Software-defined caching: Managing caches in multi-tenant data centers*". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*, **2015**: 174–181.

[19] H. S. Stone, J. Turek and J. L. Wolf. "*Optimal partitioning of cache memory*". *IEEE Transactions on computers*, **1992**, *41*(9): 1054–1068.

[20] G. E. Suh, L. Rudolph and S. Devadas. "*Dynamic partitioning of shared cache memory*". *The Journal of Supercomputing*, **2004**, *28*(1): 7–26.

[21] D. K. Tam, R. Azimi, L. B. Soares *et al.* "*RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations*". In: *ACM SIGARCH Computer Architecture News*, **2009**: 121–132.

[22] C. A. Waldspurger. "*Memory resource management in VMware ESX server*". *ACM SIGOPS Operating Systems Review*, **2002**, *36*(SI): 181–194.

[23] C. A. Waldspurger, N. Park, A. T. Garthwaite *et al.* "*Efficient MRC Construction with SHARDS.*" In: *FAST*, **2015**: 95–110.

[24] X. Wang, Y. Li, Y. Luo *et al.* "*Optimal footprint symbiosis in shared cache*". In: *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, **2015**: 412–422.

[25] Z. Wang, X. Wang, F. Hou *et al.* "*Dynamic Memory Balancing for Virtualization*". *Acm Transactions on Architecture & Code Optimization*, **2016**, *13*(1): 2.

[26] J. Wires, S. Ingram, Z. Drudi *et al.* "*Characterizing Storage Workloads with Counter Stacks.*" In: *OSDI*, **2014**: 335–349.

[27] X. Xiang, B. Bao, C. Ding *et al.* "*Linear-time modeling of program working set in shared cache*". In: *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, **2011**: 350–360.

[28] T. Yang, E. D. Berger, S. F. Kaplan *et al.* "*CRAMM: Virtual memory support for garbage-collected applications*". In: *Proceedings of the 7th symposium on Operating systems design and implementation*, **2006**: 103–116.

[29] W. Zhao, Z. Wang and Y. Luo. "*Dynamic memory balancing for virtual machines*". *ACM SIGOPS Operating Systems Review*, **2009**, *43*(3): 37–47.

[30] P. Zhou, V. Pandey, J. Sundaresan *et al.* "*Dynamic tracking of page miss ratio curve for memory management*". **2004**, *32*(11): 177–188.

[31] Y. Zhou, J. Philbin and K. Li. "*The Multi-Queue Replacement Algorithm for Second Level Buffer Caches.*" In: *USENIX Annual Technical Conference, General Track*, **2001**: 91–104.

致谢

在毕业论文的写作过程中我需要感谢我的导师，汪小林、罗英伟以及王振林老师，虚拟化内存工作集预测涉及非常多的系统底层知识：操作系统，虚拟化等等，只是书本上的知识是远远不够的，老师们的经验和意见给了我极大的启发和帮助。通过在做实验不断验证过程中和老师们的沟通加深了对问题本质的理解。尽管在完成这个系统的时候遇到了极大的障碍并且在最开始的时候走错了方向，不过经过跟老师们的深入讨论，后来从另外一个方向切入并且找到了论文突破口。

我还要感谢我的大师兄王志刚，在刚进入实验室的时候我对于虚拟化和操作系统的认识都是肤浅的，王志刚给予我了很大的帮助，帮助我能够快速吸收虚拟化的知识，通过指导我做一些小实验来理解和认识实验室使用的虚拟化环境。王志刚不仅给我启蒙教育，我们还一块儿合作完成了内存工作集预测和调度的期刊文章，并且对于我这篇用 AET 做内存工作集预测的工作起到了指导和参考作用。

另外还要感谢实验室的众多同学，胡静远，白晓旷，胡夏蒙，周岚。我们都在做相关性比较强的工作，所以我们经常能够深入讨论问题，特别是 AET 算法的作者胡夏蒙和周岚，我们一同探讨了 AET 算法在内存工作集预测上的可行性。由于做系统底层研究的同学本来就很少，所以很庆幸还能和实验室的同学深入交流系统底层知识，在交流中也丰富了我的认识，让我的理解更加深刻。

最后我要感谢我的父母以及我的女朋友孔颖，他们在生活上给了我很多帮助，包括在写作论文任务最重的四五月份我还发烧，是我的女朋友陪我打点滴，在医院边打点滴边写论文，这些经历都是我人生宝贵的财富。

我的这篇工作只是虚拟化内存预测调度的一点很小的贡献，虚拟化内存预测调度仍然是一个非常有难度和挑战的工作，希望实验室未来能够在这个方向更加深入，再接再厉，发出更加优质的文章。