基于内存迭代拷贝的Xen虚拟机动态迁移机制研究

熊安萍 徐晓龙
(重庆邮电大学计算机科学与技术学院 重庆 400065)

摘 要 为减少虚拟机动态迁移时间,Xen采用了Pre-Copy算法来选择合适的时间进行停机拷贝,以保证在低负载或空负载时的虚拟机迁移的优越性能,但在高负载环境下,Pre-Copy算法对内存页的重复迭代严重影响了虚拟机迁移的效率。基于Xen虚拟机内存迭代拷贝算法,提出了一种内存分片迭代拷贝机制,即通过缩短迭代拷贝的终止时间来减少虚拟机动态迁移所花费的时间。实验结果表明,内存分片迭代机制可以有效提升Xen虚拟机动态迁移的性能。

关键词 动态迁移,虚拟机,Pre-Copy算法,分片迭代拷贝,迁移时间
中图法分类号 TP393 **文献标识码** A

Research on Mechanism of Xen Virtual Machine Dynamic Migration Based on Memory Iterative Copies

XIONG An-ping XU Xiao-long
(College of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing 400065, China)

Abstract In order to reduce the dynamic migration time of virtual machine, the pre-copy algorithm was used in Xen virtual to choose the right time for downtime copy to guarantee the superior performance of virtual machine's migration in low load or empty load. But with heavy load, Pre-copy algorithm needs to iterative repeatly to the memory pages, which affects processing performance seriously. With algorithm of Xen virtual machine iterative copies, we proposed a mechanism of fragments interative copies to reduce virtual machine live migration time by cutting the termination of time to iterative copy. The results of experiment show that mechanism of memory fragment iteration enhances performance of the live migration of Xen virtual machine.

Keywords Dynamic migration, Virtual machine, Pre-Copy algorithm, Fragments interative copies, Migration time

虚拟化[1]作为云计算的关键技术之一,其数据中心的服务器硬件物理资源虚拟成多个虚拟资源[2],每个虚拟化资源都可以作为运行主机,且保证高独立性,以实现对数据的高速并行处理,大大提高硬件资源的使用效率[3]。虚拟机动态迁移是保障整个虚拟化平台可靠运行的关键技术,能够实现自动的服务器维护以及不停机的对用户透明的实时迁移。虚拟机动态迁移对象主要包括存储、网络、CPU和内存。其中CPU状态和I/O状态固然重要,但是它们只占迁移总数据量很少的一部分,封装和传送都比较方便;硬盘资源虽然数据量非常大,但在局域网内可以通过NFS的方式来实现;网络资源可以通过发送ARP重定向包,将VM的IP地址与目的服务器的MAC地址相绑定,用以实现网络的无缝迁移;内存的迁移是整个动态迁移中难度最高的[4],因为内存中的信息不仅数据量比较大,而且会时刻发生变化。

Xen[5]是一个开放源代码的虚拟机系统软件,由剑桥大学开发。其虚拟机性能与真实机器性能相差3%左右[6]。Xen目前被广泛地用于服务器应用整合、软件开发测试和集群系统应用等方面。本文针对Xen虚拟机动态迁移机制中预拷贝迭代算法的缺点和不足,提出了一种分片的内存迭代拷贝机制,用以优化虚拟机动态迁移的时间开销,提高系统动态迁移的性能。

1 相关研究

近年来,国外对虚拟机动态迁移技术的研究主要集中在工业界和产业界[7]。虚拟机动态迁移技术可以分为预拷贝迁移方法、后续拷贝迁移方法。预拷贝迁移方法在基于hypervisor的虚拟机系统中已得到应用,如VMware、Xen、KVM等。后续拷贝迁移方法最初在进程迁移中被提出,有较早的使用文件服务器实现的Freeze Free,其后也有人提出在openMosix内核中实现后续迁移,最近在SnowFlock项目中实现了操作系统级的后续迁移。

国内学者对Xen动态迁移技术给予了广泛的关注和研究。李建彬针对Xen动态迁移技术应用于虚拟机的分布式容灾备份领域的不足,提出了DRO动态迁移框架和HIT动态迁移算法,以及脏页减速器,从而减少了总迁移时间和宕机时间[8];阮敏针对减少Xen动态迁移时间,提出了分层拷贝

到稿日期:2012-10-15 返修日期:2013-02-26 本文受重庆市教委科学技术研究项目(KJ120513),工信部2012年物联网发展专项资金项目(2-5)资助。

熊安萍(1970-),女,副教授,CCF会员,主要研究方向为分布式计算、操作系统内核,E-mail:xiongap@cqupt.edu.cn;徐晓龙(1988-),男,硕士生,主要研究方向为虚拟化技术。

算法和脏页减速算法[8]；高翔针对 Xen 动态迁移的传统迭代过程中重复拷贝相同脏页的问题，提出了页面分层算法[9]；付超针对 Xen 动态迁移在内存页面频繁更改时所体现出来的传送效率低下的问题，提出了基于 Excalibur 系统的分层矩阵拷贝算法[10]。这些算法虽然在一定程度上优化了 Xen 的动态迁移性能，但是也需要相当大的资源消耗。

2 Xen 动态迁移机制

Xen 虚拟机动态迁移时间包括迭代拷贝时间及停机拷贝时间，其中，迭代拷贝时间是指从预拷贝的第一次迭代开始到整个迭代结束花费的时间，停机拷贝时间是指虚拟机真正停止工作的时间。

2.1 Xen 动态迁移

本文把源物理计算机记为主机 A，目的物理计算机记为主机 B，虚拟机记为 VM。Xen 动态迁移采用的预拷贝方法的主要步骤如图 1 所示。

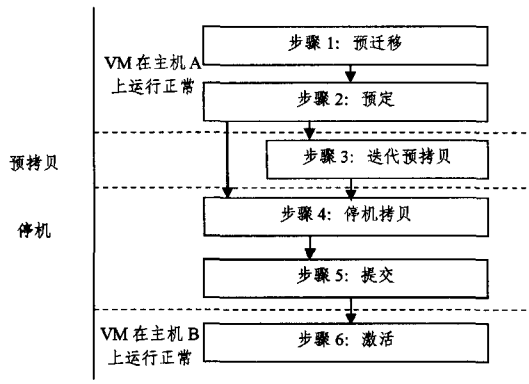

图 1 Xen 动态迁移步骤

源主机 A 要迁移其上的一个虚拟机 VM，则首先选择一个目的计算机作为 VM 的新主机，通过 socket 向主机 B 发起动态迁移的请求；先检测 B 的资源是否满足需求，若满足，则向 B 预定资源，若不满足，则选择其他主机作为 VM 的新主机；接着，主机 A 以迭代的方式将 VM 的内存页拷贝到主机 B 上，每轮只拷贝前一轮传送过程中被修改过的页面，直到迭代拷贝结束；之后进行停机拷贝，即主机 A 上的 VM 把网络连接重定向到 B，CPU 状态和前一轮传送过程中修改过的页都在这个步骤被传送；主机 A 收到了主机 B 成功接收 VM 的映像的消息，停止运行其上的 VM；最后，启动已经迁移到 B 上的 VM，迁移后使用 B 的设备驱动，广播新的 IP 地址。至此，完成整个虚拟机的动态迁移。

2.2 Xen 动态迁移内存迭代拷贝策略

由以上迁移过程可以看出，影响 Xen 动态迁移效率的在于核心的内存迭代拷贝算法，该算法的主要步骤如下：

(1)第一轮拷贝所有的内存页；

(2)第二轮拷贝在第一轮迭代过程中修改过但在本轮并没有被修改的内存页；

(3)以此类推，第 n 轮拷贝的是在第 $n-1$ 轮迭代过程中修改过但在本轮没有修改的页。且每轮迭代结束后，判断迭代次数是否达到定义的最大迭代次数(默认设定为 30)或某轮迭代过程中被修改的页数是否小于 50，如果满足就进入停机拷贝步骤，把源服务器上剩余的被修改的内存页都拷贝到目的服务器上；如果不满足，则继续迭代。

迭代拷贝算法与静态迁移算法相比较，优势明显，迁移时间明显少于静态迁移，且对用户是透明的。在低负载的情况下，这种内存拷贝算法的停机时间可以保证在 0.2s～0.5s 之内，这已经是一个用户几乎感觉不到的时间了。但是缺点也是比较明显的，尤其是在高负载情况下，内存页的动态变化非常频繁，导致动态迁移过程中迭代拷贝时间延长，迁移性能下降。

3 基于内存分片迭代拷贝的迁移机制

从 Xen 迁移机制分析可知，进行高效的虚拟机动态迁移，关键是要找到一个最佳的停止迭代拷贝并进行停机拷贝的时机，以缩短虚拟机动态迁移过程中内存迭代拷贝的时间及停机拷贝时间，从而提升虚拟机迁移效率。本文以缩短内存迭代拷贝时间为目标，提出了一种分片的内存迭代拷贝机制。

3.1 内存分片的迭代拷贝机制

通过研究动态迁移机制，本文定义虚拟机动态迁移过程中系统确定的停止内存迭代拷贝的固定的脏页数称为额定脏页数，即迭代算法执行直到内存脏页数少于这个固定脏页数停止。要缩短迭代拷贝的时间，就要找到最快到达迭代拷贝停止的点[11]，这样可以较快结束迭代拷贝，如图 2 所示。

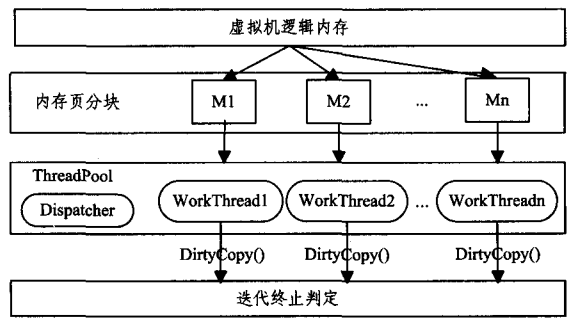

图 2 内存分片迭代机制

算法的核心思想是将内存页划分为若干块，本文定义块的大小为 64M，如果虚拟机分配了 128M 物理内存，则可划分为 2 块，以此类推。虚拟机动态迁移进行内存迭代拷贝时，分别由独立的线程来完成内存迭代拷贝，让每轮迭代过程中产生的脏页数小于额定脏页数的概率变大，从而有效提升迁移效率。

定义 3 种页位图来区分 Xen 动态迁移过程中内存页的修改状态：to_send、to_skip、to_fix，其分别表示上轮迭代过程中出现的脏页、本轮迭代过程中出现的脏页、停机拷贝阶段传送的页。

基于内存分片迭代机制的核心算法步骤如下：

(1)建立线程池，包括一个分发线程和多个工作线程；

(2)内存分块，根据内存大小分成相应块数，并记录每块内存页的最小 page_min 和最大编码 page_max；

(3)变量初始化，包括迭代次数 iter=0、发送过的脏页数 sent_this_iter=0 以及是否最后一轮标识 last_iter=0 等；

(4)开始一轮迭代，即分发线程，唤醒工作线程，每个工作线程调用 DirtyCopy()函数，该函数实现脏页检查和拷贝，描述为：

```
内存页编号=page_min
while(内存页编号<page_max){
```

```
if(! last_iter){
  调用 xc_shadow_control()函数将本轮目前的脏页拷贝到 to_
  skip 中;
}
for(;内存页<page_max;内存页++){
  if(last_iter OR to_send 为 1 且 to_skip 为 0){
    把该页放入缓冲区 buffer;
  }
  if(buffer 已满){
    传送 buffer 中的内存页;
  }}}
```

(5)如果迭代次数达到 30 次或者本轮迭代页面与本轮被修改页面之和小于 50,则将 last_iter 置为 1,进入最后一轮,即停机拷贝阶段,否则重复执行第(4)步。

3.2 内存分片迭代机制的理论分析

设某个迁移时刻内存初始迭代集为 U,假设迭代拷贝期间系统负载基本稳定,脏页的生产率呈平均分布,脏页产生的平均概率为 P,内存页拷贝速率为 v,分块后的迭代子集为 $u_1, u_2, \cdots u_n$,则

$$u_1 = u_2 = \cdots = u_n = U/n \tag{1}$$

原迭代算法第一轮迭代所需的时间为

$$T_1 = U/v \tag{2}$$

对内存进行分块并采用多线程迭代后,每个内存块第一轮迭代所需的时间为

$$t_1 = u_1/v \tag{3}$$

因为 $u_1 < U$,所以由式(2)和式(3)推出

$$t_1 < T_1 \tag{4}$$

对于第二轮,原有迭代拷贝算法的迭代集变为

$$U' = U * P * T_1 \tag{5}$$

而分片迭代算法第二轮迭代集为

$$u_1' = u_1 * P * t_1 \tag{6}$$

由式(1)和式(4)推出 $u_1' < U'$。

以此类推,每轮的迭代集都会变小,则迭代拷贝所需时间也必然变少,因此可以使迭代的停止时间提前。另外,由于每轮产生的脏页集在变小,这样也会促使迭代过程中产生的脏页数小于额定脏页数的概率变大。

因此,内存分片迭代机制会加快迭代拷贝的停止,减少迁移时间。

4 实验及分析

4.1 实验环境及方法

实验搭建 hostA、hostB、hostC 物理主机,配置为 4G 内存、500G 外存,操作系统为 CentOS5.5,具体如下:

(1)hostA:linux NFS 服务器,通过网络为 XenDomain 提供存储空间。

(2)hostB:Xen 主机服务器,即源服务器,它使用 NFS 导出的目录运行一个客户机 Domain。

(3)hostC:Xen 主机服务器,即目的服务器,它是来自 hostB 服务器的客户机 Domain 的迁移目的地。

Xen 动态迁移要求共享存储器,所以创建了一个 NFS 服务器,欲迁移的 Domain 使用的就是 NFS 服务器上的共享存储空间。源服务器上创建了 3 个虚拟机,内存分别为 256M、512M 和 1024M。分别采用原有的迭代拷贝算法及本文的分片迭代拷贝机制,进行如下实验过程:

首先在零负载的情况下进行迁移,记录迭代次数、停机拷贝时间以及总迁移时间的数据,这样测试 20 组,计算出平均值。再删除这 3 个虚拟机,创建内存分别为 128M 和 2048M 的 2 个虚拟机,进行同样的迁移测试。接着模拟一种高负载的情况,具体方法是在动态迁移的同时编译一个 Linux 内核,这样会使源服务器的 CPU 在虚拟机动态迁移阶段内一直很繁忙。分别进行 20 组 5 种配置的虚拟机动态迁移的实验。

4.2 实验结果及分析

在零负载环境下,原有迭代算法及分片迭代算法的实验数据如表 1 和表 2 所列。

表 1 零负载 Pre-copy 算法动态迁移结果

内存(M)	迭代次数	停机时间(s)	总迁移时间(s)
128	5	0.391	11.825
256	5	0.409	25.552
512	4	0.411	49.928
1024	4	0.400	56.613
2048	4	0.397	66.274

表 2 零负载基于分片迭代算法动态迁移结果

内存(M)	迭代次数	停机时间(s)	总迁移时间(s)
128	3	0.388	7.24.8
256	3	0.398	19.632
512	3	0.393	37.245
1024	4	0.397	48.356
2048	4	0.396	59.685

零负载情况下的实验结果对比如图 3 和图 4 所示。

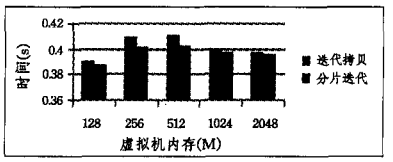

图 3 零负载时停机拷贝时间比对

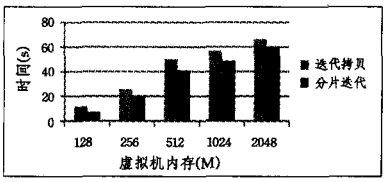

图 4 零负载时总迁移时间比对

由以上实验结果可知,在零负载环境下,基于内存分片迭代拷贝机制与原有的迭代机制相比,停机拷贝时间较为接近,内存分片机制的总迁移时间也会由于总的停止迭代拷贝的时间缩短而有一定的减少。

在高负载环境下,原有迭代算法及分片迭代算法的实验数据如表 3 和表 4 所列。

表 3 高负载动态迁移结果

内存(M)	迭代次数	停机时间(s)	总迁移时间(s)
128	30	4.102	42.963
256	30	4.015	61.829
512	30	4.149	89.531
1024	30	4.155	96.238
2048	30	4.652	117.212

(下转第 99 页)

级方案有一个接近于 $k\log k$ 因子的优化。

结束语 共谋密钥攻击是加密广播业务中一种常见的攻击。本文提出了一种有效对抗该攻击的新方案，并基于加密广播技术和 Hash 函数理论，构建了三级结构的密钥方案、加密方案、解密方案和三级叛徒追踪算法。在利用 Chernoff 界论证方案安全性的过程中，给出了合理的系统参数值。对比已有的 CFN 对称方案，本文方案显著优化了 3 个主要性能参数：个人密钥存储复杂度、数据冗余复杂度和个人解密盒计算复杂度。新方案的构造方法具有较高的理论参考价值。利用具体的 Hash 函数和子密钥值实例化本方案，可以构建一个应用型的加密广播解决方案。

考虑到系统端的实际计算能力，多级方案中的级数必定存在实际上限。如何取得最优的实际上限是个值得深入研究的课题。

参 考 文 献

[1] Fiat A, Naor M. Broadcast encryption [A]// Advance in Cryptology-CRYPTO'93 [C]. vol. 773, Berlin, Germany: Springer-Verlag, 1994: 480-491

[2] Chor B, Fiat A, Naor M. Tracing traitors [A] // Advance in Cryptology-CRYPTO'94 [C]. vol. 839, Berlin, Germany: Springer-Verlag, 1994: 257-270

[3] Naor M, Pinkas B. Threshold traitor tracing [A]// CRYPTO'98 [C]. vol. 1462, Berlin: Springer-Verlag, 1998: 502-517

[4] Kurosawa K, Desmedt Y. Optimum traitor tracing and asymmetric scheme [A] // Advances in Cryptology-EUROCRYPT'98 [C]. vol. 1403, Berlin: Springer-Verlag, 1998: 145-157

[5] Boneh D, Franklin M. An efficient public key traitor tracing scheme [A] // CRYPT'99 [C]. vol. 1666, Berlin, Germany: Springer-Verlag, 1999: 338-353

[6] Fiat A, Tassa T. Dynamic traitor tracing [A]// Advance in Cryptology-CRYPTO'99 [C]. vol. 1999, Berlin, Germany: Springer-Verlag, 1666: 354-371

[7] Safavi-Naini R, Wang Ye-jing. Squential traitor tracing [A]// Advance in Cryptology-CRYPTO2000 [C]. vol. 1880, Berlin, Germany: Springer-Verlag, 2000: 316-332

[8] Goldreich O, Goldwasser S, Micali S. How to construct random functions [J]. J. Assoc. Comput, 1986, 33(4): 792-807

[9] Mu Ning-bo, Hu Yu-pu, Ou Hai-wen. Broadcast encryption schemes based on RSA [J]. The Journal of China Universities of Posts and Telecommunications, 2009, 16(1): 69-75

[10] Wu Yong-dong, Deng R H. A Multi-Key Pirate Decoder against Traitor Tracing Schemes [J]. Journal of Computer Science and Technology, 2010, 25(2): 362-374

(上接第 65 页)

表 4 基于分片迭代算法高负载动态迁移结果

内存(M)	迭代次数	停机时间(s)	总迁移时间(s)
128	30	3.917	31.296
256	30	3.756	44.284
512	29	3.872	62.246
1024	30	4.039	70.012
2048	29	4.078	91.259

高负载情况下的实验结果对比如图 5 和图 6 所示。

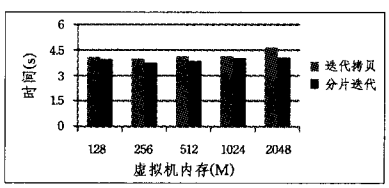

图 5 高负载时停机拷贝时间比对

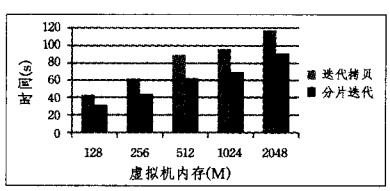

图 6 高负载时总迁移时间比对

在高负载情况下，比较而言，基于内存分片迭代拷贝机制不仅停机拷贝时间有一定的减少，而且总迁移时间也减少得较为明显，这是因为分片的迭代拷贝机制迭代过程中产生的脏页数小于额定脏页数的概率变大，从而加快了迭代拷贝的终止。

综上所述，在零负载或低负载情况下，基于分片内存迭代拷贝机制保持了原有 Pre-Copy 算法的性能，总的迁移时间略有改善；但在高负载情况下，内存分片迭代机制优越性明显，提升了虚拟机动态迁移的效率。

结束语 本文分析了 Xen 动态迁移中的内存迭代拷贝机制，提出了基于分片迭代的动态迁移机制，通过分析和实验验证了系统在不同负载环境下的迁移性能。下一步将继续优化 Xen 的动态迁移，将分片迭代拷贝中修改过于频繁的内存页直接放到停机拷贝的脏页中，提出缺页错误检测机制，以解决虚拟机迁移时因停机拷贝过长而导致缺页的问题。

参 考 文 献

[1] 刘铭，张炯，龙翔. 基于 Xen 虚拟机全系统在线增量迁移的设计与实现[J]. 微计算机信息，2010，26(10-3)：162-164

[2] Intel Corporation. System virtualization: principle & implementation[M]. Beijing: Tsinghua University Press, 2008

[3] 董耀祖，周正伟. 基于 X86 架构的系统虚拟机技术与应用[J]. 计算机工程，2006，32(13)：71-73

[4] 刘鹏程，陈榕. 面向云计算的虚拟机动态迁移框架[J]. 计算机工程，2010，36(5)：37-39

[5] 刘可超，李小勇. 基于 Xen 的虚拟磁盘调度算法改进[J]. 微型电脑应用，2010，26(4)：51-53

[6] Barham P, Dragovic B, Fraser K, et al. Xen and the art of virtualization[C]// Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP19). New York: ACM Press, 2003: 164-177

[7] 阮敏. Xen 环境下实时迁移结构和算法研究[D]. 大连：大连海事大学，2009

[8] 李建彬. 基于虚拟机的分布式容灾备份技术研究[D]. 长沙：国防科学技术大学，2010

[9] 高翔. 基于 Xen 的虚拟机动态迁移算法优化[D]. 深圳：哈尔滨工业大学，2010

[10] 付超. 基于 Excalibur 系统的 Xen 动态迁移优化与研究[D]. 北京：北京邮电大学，2011

[11] Clark C, Fraser K, Hand S, et al. Live migration of virtual machines[C]// Proceedings of the 2nd Conference on Networked Systems Design & Implementation. 2005: 273-286