全相连TLB 替换算法的比较

刘华平[1] 韩承德[2]

[1,2](中国科学院 计算技术研究所 系统结构室,北京 100080)

E-mail: lhp@ict.ac.cn

内容摘要

本文简要介绍 TLB 的管理方法，详尽介绍了三种全相连 TLB 结构替换算法的实现：随机替换算法，LRU 算法和本文提出的简化 LRU 算法。同时，本文对这三种算法就访问 TLB 的失效率以及实现这三种算法的硬件复杂度进行全面比较。通过实验，我们得出与随机算法相比，LRU 算法和简化 LRU 算法对访问 TLB 失效率的改进效果相当，大约有 5.48%的增益。同时我们分析得出，与随机算法相比，简化 LRU 算法只需要增加较少硬件就能减少 5.48%访问失效率，与 LRU 算法相比，简化 LRU 算法需要的硬件较简单，同时不会引起关键路径时间的增加。

关键词：TLB，随机替换算法，LRU（Least Recently Used）算法

1．背景介绍：

TLBS（ translation lookaside buffers ） 作为实现虚实地址转换的必要部件已经广泛使用到当代通用处理器中（1， 4， 5）。在这些的处理器中，TLB管理的实现方法一般分为两种，一种是硬件直接管理TLB部件，即对TLB部件的管理全部由硬件实现；另一种是通过软件来管理TLB部件，即增加特定的指令，和操作系统一起实现对TLB的管理。这两种方法各有各的优缺点：首先，对于硬件管理TLB的方法来说，在缺页时或页失效时，替换或填入TLB项可以减少很多时间，但是在硬件实现需要付出很大的代价；而对于软件管理TLB的方法（7），在缺页时或页失效时，替换或填入TLB项时虽然要较多时间，但是大大减少硬件实现和控制的复杂度。在本文中，我们讨论对象是软件管理TLB的方法。

TLB组织方式一般分为，全相连结构；组相连结构；直接相连结构。各种结构有各的优缺点。在本文中，我们不对组相连和直接相连的结构讨论，我们讨论对象是全相连结构的TLB。

在很多的软件管理TLB的处理器中，为了减小管理TLB的硬件复杂度，当访问TLB不命中或失效时，替换TLB的方法做得比较简单。在MIPS处理器中（2），采用了随机替换的方法，即替换TLB项时根据一个随机数来替换TLB的内容。产生随机数的方法非常简单，硬件的实现方法相应也比较简单，但是这种方法有一种缺点，就是它替换有用TLB表项的概率较大，这样会降低访问TLB的命中率。因为换出的TLB表项，有可能马上要被访问，从而又重新换入TLB。上述的TLB替换方法虽然硬件实现比较简单，但是，因为替换TLB的方法比较简单，从而导致访问TLB的命中率较低，影响处理器性的性能。

随着ASIC技术的发展，处理器硬件可以做得越来越复杂。当TLB部件不是关键路径时，如果选择一种较好的替换算法能提高TLB访问的命中率，同时实现的硬件复杂度增加不是太大，这将是一种比较可取的方法。在本文中，将以全相连的TLB为对象，提出使用简化的LRU算法来管理TLB

作者简介： 刘华平（1977-），男，江西永新人，博士生，主要研究领域计算机系统结构；韩承德，研究员，博士生导师，计算所首席科学家，主要研究领域分布式处理系统，并行处理系统，计算机系统结构

的实现方法，并与随机替换方法和传统的LRU算法进行对比。

本文的组织方法如下：第二节，简单介绍全相连TLB的管理方法；第三节，介绍TLB随机替换算法的实现；第四节，详尽介绍传统的LRU替换算法应用到管理TLB替换的实现；第五节，详尽介绍本文提出的简化LRU替换算法管理TLB的实现；第六节，给出我们实验环境；第七节，将随机替换算法，LRU算法，和简化的LRU算法实现进行性能比较，以及硬件复杂度的比较；第八节，得出结论。

2．TLB 的软件管理方法

当访存指令的虚地址访问 TLB 不命中，或虚地址对应的 TLB 项无效时，需要把内存中页表项中的 TLB 项换入 TLB 中。这是因为程序的局部性使得该指令后的访存指令很有可能访问同一页的数据，也就是说之后的指令很有可能还要通过该 TLB 项内容完成虚地址到实地址的变换。由于硬件存放的 TLB 项数量是有限的，换入 TLB 项的同时，硬件中的 TLB 项必须换出 TLB。

选择换出的项和 TLB 的组织方式有很大关系。在直接相连的 TLB 中，因为 TLB 项和该项数一一对应，所以选择换出的 TLB 项只有一种选择；在组相连的 TLB 中，要换进的 TLB 和一组（每组包括的 TLB 个数由组相连决定，两组相连则每组包括两项）对应，供换出 TLB 的选择是组内的 TLB 项。而对于全相连组织的 TLB 来说，可以换出 TLB 中的任何一项，如何很好选择换出的 TLB 项对性能影响较大。因为换出 TLB 的不确定，很有可能换出 TLB 的项正好是将要使用的 TLB 项，如果这种情况发生的概率很大，将大大降低访问 TLB 的命中率，从而会大大降低处理器的整体性能。下面，我们将介绍全相连 TLB 结构的替换算法。

3．TLB 随机替换算法：

当虚地址访问 TLB 不命中时，或虚地址对应的 TLB 项无效时，处理器需要把 TLB 其中一项换出 TLB，随机替换算法是处理器根据自动生成的随机数决定换出的 TLB 项。以 MIPS R4000 处理器为例（2），MIPS R4000 处理器采用的是有序发射，乱序执行，有序结束的结构。该处理器为实现 TLB 替换设置一个寄存器，该寄存器存放将被替换出的 TLB 项对应的随机数。而随机数产生规则如下：当初始化处理器时，给该寄存器赋 TLB 项数的最大值，如果 TLB 总共有 48 项（从 0 算起，项数最大值为 47），则给该寄存器赋初始化值 47。在处理器运行过程中，当处理器成功结束一条指令时，给存放该随机数的寄存器值循环减 1，即当该寄存器的值等于 TLB 项的最小值（即 0）时，该寄存器的值又回到 TLB 项的最大值（47）。当访问 TLB 失效时，需要把内存中页表的某项换入 TLB，处理器就根据该寄存器值把需要的页表项填入该寄存器对应 TLB 项中。

据上述随机替换算法，不难知道该方法有比较大的缺点，即该方法使得替换出 TLB 的项是即将要使用的 TLB 表项的概率比较大。如果这种情况发生，当程序运行访存虚地址恰好影射到刚换出 TLB 表项时的指令，处理器就必需把刚换出 TLB 的项重新填入 TLB。我们知道，当处理器访问 TLB 失效时，处理器必须停止运行所有的程序，同时必须取消已经取出的指令，然后发出 TLB 失效中断，最后由操作系统处理，这个访问 TLB 不命中处理过程要使用很长的时间。如果，频频发生替换出 TLB 的想是即将要使用的 TLB 项时，会大大降低访问 TLB 的命中率。所以我们可以改进随机替换算法，减小上述情况出现的概率，从而提高处理器的性能。

4．LRU 替换算法

与随机替换算法相比较，我们使用 LRU 算法可以减小把即将使用的 TLB 项换出的概率，提

高 TLB 访问的命中率。

LRU 替换算法基本原理是在访问 TLB 失效时，选择被换出的 TLB 项是根据每个 TLB 被访问时间离当前时间的长短来决定的，时间最长的项认为是最不可能将要使用的项，作为被换出的对象。

在具体的实现中，我们为每项 TLB 设置一个计数器，每次根据每个计数器的值来选择被替换的项，数值最小的被选为被替换的对象。 设计的计数器能存放的值越大越好，因为访问同一个 TLB 项的指令较多。但为了节省硬件资源，可结合实际操作系统对进程的管理来确定。在我们的设计中，我们把每个计数器设计成 32 位，能计的最大值为 4296947295（在我们的程序中访问 TLB 的指令不会超过该数）。基本实现过程如下：在初始化状态，我们给每个计数器都赋初值 0；在 TLB 刚被访问时，我们给该 TLB 项对应的计数器赋最大值，即 4296947295，同时给其他 TLB 对应的每个大于 0 的计数器减 1。在选择被替换的 TLB 项时，我们查找每个 TLB 项对应计数器的值，如果计数器值是最小，则该 TLB 就是被替换的对象。如果有多个最小值相同的计数器时，我们选择项数最小的 TLB 作为被替换的对象，如第 1 项和第 8 项 TLB 对应的计数器同时为最小值，即选择第 1 项做为替换对象。基本的算法如下：

```
VAR=第一项 TLB 的计数器值;
for  (I=0;I<TLB 的总数; I++) {
  if  (第 I 项 TLB 的计数器值<VAR) {
    VAR=第 I 项 TLB 的计数器值;
     J=I;
  }
}
被替换的 TLB 项寄存器值= J ;

initial
  for  (I=0;I<TLB 的总数; I++)
    TLB 对应的计数器[I]=0;

if  (当 TLB 被访问时) {
  for  (i=0;I<TLB 的总数; I++)
       if  (TLB 对应的计数器[I]>0)  TLB 对应的计数器[I]-- ;
  被访问的 TLB 项对应的计数器= 设定的最大值（在本设计中为 4296947295） ;
}
```

TLB 的 LRU 替换算法描述

与随机替换算法相比较，实现LRU算法需要更多的寄存器（在本设计中，每项TLB要增加一个32位的计数器），同时控制电路也很复杂。LRU总是选择最长时间没有访问的TLB项作为替换对象，虽然这种算法也有可能把即将访问的TLB项替换出TLB，但是比随机算法把即将访问的TLB项替换出TLB的概率小。这样，可以降低访问TLB的失效率。

5．优化LRU替换算法

与随机替换算法相比较，该算法也可以减小把即将使用的 TLB 项换出的概率，提高 TLB 访问的命中率；但和 LRU 替换算法相比，不是单纯意义上的 LRU 算法，即此算法不一定是把离当前访问时间最长的 TLB 项换出 TLB。该算法把即将使用的 TLB 项换出的概率要大，但能减小硬件的复杂度。

简化的 LRU 替换算法基本原理和 LRU 替换算法相似，是在访问 TLB 失效时，选择被换出的 TLB 项是根据每个 TLB 被换入时间离当前时间的长短来决定的，时间最长的项被换出。

在该算法的具体实现中，我们也是为每项 TLB 设置一个计数器（计数器的大小根据 TLB 项数决定，假如 64 项全相连的 TLB，我们需要 6 位的计数器（2）6=64），根据每个计数器的值来选择被替换的项。 在初始化状态，我们给每个计数器都赋初值 0；在 TLB 刚被换进 TLB 时，我们给其对应的计数器赋 TLB 项数的最大值，假如 TLB 的总数为 64 时，赋值为 63，同时给其他 TLB 对应的每个大于 0 的计数器减 1。在选择被替换的 TLB 项时，我们查找每个 TLB 项对应计数器的值，如果计数器值等于 0，则该 TLB 就是被替换的对象（从我们设定的算法看，任何一个时刻，至少一个计数器的值为 0）。如果有多个等于 0 的计数器时，和 LRU 算法一样，我们也固定选择项数最小的 TLB 作为被替换的对象。基本的算法如下：

```
for  (I=0;I<TLB 的总数；I++)
   if  (TLB 对应的计数器[I]==0)  {
     被替换的 TLB 项寄存器值= I ；
     break;
   }

initial
  for  (i=0;I<TLB 的总数；I++)
    TLB 对应的计数器[I]=0;

if （页表项写入 TLB）{
  for  (i=0;I<TLB 的总数；I++)
   if  (TLB 对应的计数器[I]>0)  TLB 对应的计数器[I]-- ;
  被写的 TLB 项对应的计数器= TLB 的总数-1；
}
```

简化的 LRU 替换算法描述

与随机替换算法相比较，实现该算法复杂，但是和 LRU 算法需要较少的寄存器，同时控制电路也简单很多。在我们的设计中，我们的 TLB 有 48 项，这样只要为每项 TLB 设置一个六位的计数器。下面，我们就详尽介绍实现随机算法，LRU 算法，简化的 LRU 算法的实验环境，以及实现这些算法性能和硬件复杂度进行比较。

6． 实验环境介绍

我们讨论的对象是软件管理的全相连结构的TLB，主要目的是比较随机替换算法，LRU算法以及简化的LRU算法对访问TLB命中率的影响，同时比较使用这三种算法的处理器性能差异。

我们实验基于的处理器是有序发射，乱序执行，有序结束的单发射结构。该结构的处理器用模拟器来实现。我们实验基于的处理器的基本指标如表1：

TLB	全相连结构的 TLB 48 项
指令 CACHE	二路相连 8 K
数据 CACHE	二路相连 8 K
取指数（/周期）	1 条
有序发射	是
乱序执行	是
有序结束	是

表1： 处理器的基本指标

由于模拟器频率很低，所以这个速度使得我们不可能运行大型的测试程序。但是为了比较上述TLB替换策略的性能，我们把启动 Linux 操作系统，几个操作命令和一些测试应用程序，作为比较对象。我们 Linux 操作系统是2.4内核；操作命令包括:ls,显示当前目录下的所有文件，pwd：显示当前所在的目录；测试应用程序包括whitestone,gzip,mb。应用程序的说明如表2。

Whitestone	一个测试浮点性能的程序
Gzip	一个压缩程序
Mb	分块矩阵大小 128*128

表2

7．及硬件实现复杂度的比较

为了比较各算法的性能，我们主要比较各算法访问TLB的失效率。访问TLB失效率定义如下：

失效率=访问TLB不命中的总数 / 访问TLB的总数；

访问TLB失效将给系统带来很大的开销，特别是在软件管理TLB的机器中。在当代处理器中，有很多机器是采取软件管理TLB的方式，同时有时多发射，乱序执行的系统结构。如果一发生访问失效，那么必须取消所有在该访存操作之后已经执行的操作，然后执行由于访存失效而引起的中断处理程序，最后把需要的TLB项换入TLB中，这个过程需要花费很多个时钟周期。很显然，访问TLB失效率越大，则对性能的损失也越大。假设访存失效率提高1%，访存失效的处理过程需要10个周期，那么处理器的CPI（cycle per instruction）就得提高0.1；而如果该处理过程需要100个周期才能完成的话，那么处理器的CPI就得提高1。

实验中，在不同TLB替换算法实现的模拟器上，运行完上述设定的程序后，共产生了29个进程。实验结果中，这三个替换算法的访存总失效率，以及平均失效率如下表3。

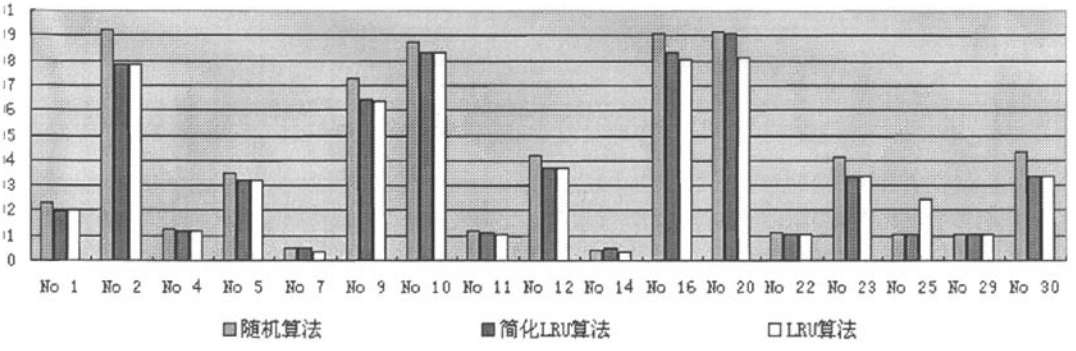
	随机替换算法	简化TLB算法	TLB算法
总失效率	0.207299673	0.195932091	0.195830422
平均失效率	0.007148265	0.006756279	0.006752773

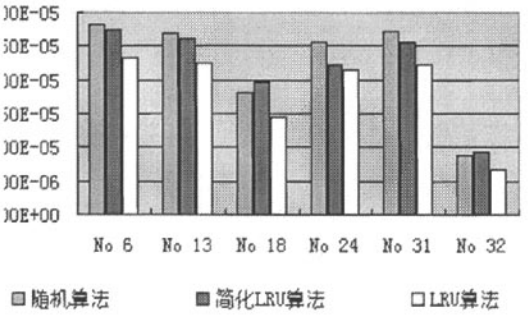
表3

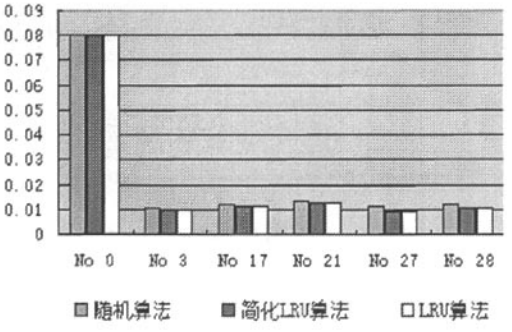
总失效率：29个进程访问TLB失效率的总和；平均失效率：总失效率除以进程总数29的商

我们从实验结果中看到，随机算法的失效率最大，而LRU算法的失效率最小。但是对总失效率而言，与随机算法相比，简化LRU算法总共降低了0.011，失效率降低了5.48%；而LRU算法和简化的LRU算法相比，没有明显的改进。

各进程访问TLB失效率如下两图。







第0，1号进程：起动 Linux 操作系统；第2号到第7号进程：执行ls命令；第8号到第14号进程：执行pwd命令；第16号到第18号：执行 Whitestone 应用；第19号到第21号进程：执行 Gzip 应用；第22号到32号进程：执行分块矩阵乘

从每个进程访问TLB失效率的比较图中，我们除第25号进程之外，LRU算法的失效率比随机算法好；而除第18号进程之外，对大部分进程而言，简化LRU算法比随机算法要好，并且效果要较明显；而LRU算法和简化的LRU算法相比，虽然有些进程的失效率有所改善，但是效果并不明显。

从上述的三种算法中，我们知道随机算法的实现最简单，随机数的产生只是反复循环的减1，且随机数最大值等于TLB项数的最大值。在本设计中，该部件只是六位的减1部件，该部件的实现也很简单，每位的表达式如下(因为本处理器中共有48项TLB，所以随机数的最大值为47)：

```
a1=!a1;
```

a2=(a1&&a2)|(!a1&&!a2);

a3=((a1|a2)&a3)|(!a1&&!a2&&!a3);

a4=((a1|a2|a3)&a4)|(!a1&&!a2&&!a3&&!a4);

a5=((a1|a2|a3|a4)&a5)|(!a1&&!a2&&!a3&&!a4&&!a5&&a6);

a6=((a1|a2|a3|a4|a5)&a6)|(!a1&&!a2&&!a3&&!a4&&!a5&&!a6);

而实现TLB算法中，需要48个寄存器来计数，48套减1部件，并且需要一个比较48个计数器数值的电路来决定其中的最小值，用于选取被替换的TLB项。如果计数器的位数比较小，电路还比较简单。而在我们的设计中，计数器是32位，那么要比随机算法增加48*32位寄存器，48套32位减1部件，还要添加一个确定48个32位计数器最小值的比较电路。这些硬件实现都比较复杂，并要增加处理器的面积，同时增加的硬件逻辑有可能加大TLB的关键路径。如果访问TLB的过程正好是处理器的关键路径，那就会降低处理器的主频。

与LRU算法相比，简化的LRU算法中计数器只要6位，所以减1电路简单很多，同时需要的寄存器也很少；另外简化的LRU算法中比较电路也简单很多，只要每个电路跟0比较就可以了。该算法因为是在写TLB项时，改变计数器的值，而写TLB项在实践中一般不会成为关键路径，所以不会增加访存的关键路径，当然也不会降低处理器的主频。

8．总结

从实验中，我们可以看出，与随机算法相比，LRU算法和简化的LRU算法能降低5.48%的失效率；而LRU算法比简化的LRU算法相比没有明显的增益。假如处理器处理访问TLB失效需要30个周期的话，和随机算法相比，LRU算法和简化的LRU算法能把CPI降低0.011。但是，要实现LRU算法需要很大的硬件复杂度，而且有可能降低处理器的主频。实现简化的LRU算法虽然也需要一定的硬件，但不是太复杂，并不会降低处理器的主频。所以在对硬件复杂度要求不太严格的情况下，本文提出的简化LRU算法比随机替换算法更适用于软件管理TLB的替换。

参考文献

1） B.L. Jacob and T.N. Mudge, "Virtual Memory in Contemporary Microprocessors," IEEE Micro, Aug. 1998,.

2）Joe Heinrich，MIPS R4000 Microprocessor User's Manual Second Edition

3）http://www.spec.org/osg/cpu2000/CINT2000/

4）B.L. Jacob and T.N. Mudge, "Virtual Memory: Issues of Implementation," Computer, Vol. 31, No. 6, June 1998, pp. 33-43.

5）B. L. Jacob and T. N. Mudge. 1998a. "A look at several memory-management units, TLB-refill mechanisms, and page table organizations." In Proc. Eighth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98), pages 295--306, San Jose CA.

6）Nick Kohout, Vinodh Cuppu, Anna Secka, Chris Collins“Performance Analysis And Design Choices For A Software Tlb ” from www.csindex.com

7）Nagle, D., Uhlig, R., Stanley, T., Sechrest, S., Mudge, T. and Brown, R. Design tradeoffs for software -managed TLBs. In Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, California, IEEE, 27-38, 1993.