处理器片上高速缓存

性能分析及优化实验

一　实验概述

（一）实验目的

1. 掌握 Cache 的基本概念、基本组织结构
2. 掌握影响 Cache 性能的三个指标
3. 了解相联度对 Cache 的影响
4. 了解块大小对 Cache 的影响
5. 了解替换算法对 Cache 的影响
6. 了解 Cache 失效的分类及组成情况
7. 了解一些基本的 Cache 性能优化方法(选做)

（二）实验步骤

1. 运行模拟器 SimpleScalar
2. 在基本配置情况下运行矩阵乘计算程序统计 Cache 失效次数，并统计三种不同类型的失效
3. 改变 Cache 容量，统计各种失效的次数，并进行分析
4. 改变 Cache 的相联度，统计各种失效的次数，并进行分析
5. 改变 Cache 块大小，统计各种失效的次数，并进行分析
6. 改变 Cache 的替换策略，统计各种失效次数，并分析
7. 对给出的矩阵乘计算程序进行适当改写以优化 cache 性能。(选做)

（三）实验平台

redhat　linux 操作系统，SimpleScalar 模拟器 。

（四）参考资料

计算机体系结构教材、SimpleScalar 模拟器使用指南等

二　背景知识

（一） Cache 基本知识

(1) 可以从三个方面改进 Cache 的性能：降低失效率、减少失效开销、减少 Cache 命中时间；

(2) 按照产生失效的原因不同，可以把 Cache 失效分为三类：

1) 强制性失效（Compulsory miss）

当第一次访问一个块时，该块不在 Cache 中，需从下一级存储器中调入 Cache，这就是强制性失效。这种失效也称为冷启动失效或首次访问失效。

2) 容量失效（Capacity miss）

如果程序执行时所需的块不能全部调入 Cache 中，则当某些块被替换后，若又重新被访问，就会发生失效。这种失效称为容量失效。

3) 冲突失效（Conflict miss）

在组相联或直接映象 Cache 中，若太多的块映象到同一组（块）中，则会出现该组中某个块被别的块替换（即使别的组或块有空闲位置），然后又被重新访问的情况。这就是发生了冲突失效。这种失效也称为碰撞失效（collision）或干扰失效（interference）。

(3) 降低 Cache 失效率的方法：增加 Cache 块大小、提高相联度、Victim Cache、伪相联 Cache、硬件预取技术、由编译器控制的预取和编译器优化。

(4) 替换算法

1) 先进先出法(FIFO)

2) 随机法：为了均匀使用一组中的各块，这种方法随机地选择被替换的块。

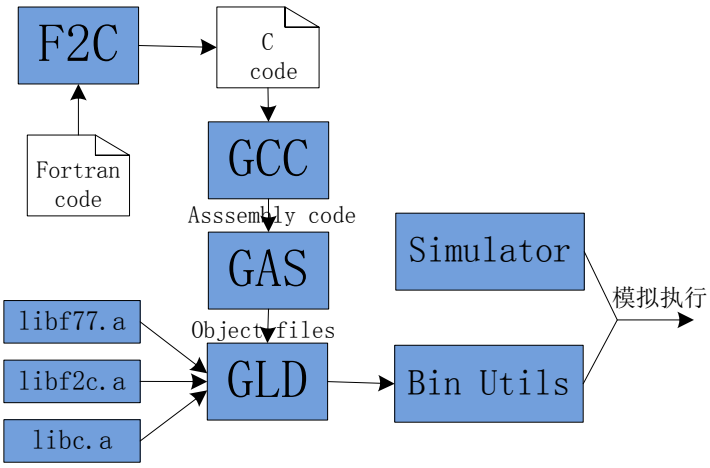
3) 最近最少使用法 LRU（Least Recently Used）：选择近期最少被访问的块作为被替换的块。但由于实现比较困难，现在实际上实现的 LRU 都只是选择最久没有被访问过的块作为被替换的块。

（二） SimpleScalar 简介及使用

SimpleScalar 是上世纪由威斯康辛大学发布的一款开源模拟器，具备良好的可移植性和可扩展性。SimpleScalar 在学术界具有十分重要的影响力，根据其官方网站统计，2000-2002 年体系结构顶级会议中有一半以上的研究者采用该模拟器来评

估他们的研究成果。

作为一款时钟精确的模拟器，SimpleScalar 采用执行驱动方式模拟，包含功能模拟和性能模拟。SimpleScalar 的指令集架构采用 C 语言宏声明，目前 3.0 版本主要支持 PISA 指令集和 Alpha 指令集。根据模拟的目的不同，SimpleScalar 包含多个模拟器实体，从最简单的 Sim-fast 到最为复杂的 Sim-outorder，可分别用于功能模拟、Cache 配置策略、流水线、前瞻预测等体系结构问题的全面研究。同时，随着 SimpleScalar 一起发布的还有一系列工具链(交叉编译器，汇编器，流水线跟踪器等)，他们与 SimpleScalar 的交互过程如图 4.1 所示。Fortran 代码需要先由 F2C 工具转换为标准 C 代码，交叉编译器则将标准 C 代码编译为模拟器后端指令集下的二进制代码，交由模拟器执行模拟，通过这些交互最终实现一个完整的模拟平台。



SimpleScalar 包含多个模拟器，复杂度由最简单的 Sim-safe 到最复杂的 Sim-outordor,下面简要介绍这些模拟器。

（1）Sim-fast

Sim-fast 是执行速度最快，最不关心模拟过程细节信息的子模拟器程序。它采用顺序执行指令的方式，没有指令并行；不支持 cache 的使用，也不进行指令正确性检查，由程序员保证每条指令的正确性；不支持模拟器本身内嵌的 Dlite！调试器（类似于 gdb 调试器）。为了模拟器的速度优化，在缺省情况下，sim-fast 模拟器不进行时间统计，不对指令的有关信息（如指令总数及访存指令数目）进行统计。当然，可以修改模拟器源程序，通过改变其设置，使模拟器更加符合设计人员的需求。

（2）Sim-safe

在工具集中，是最简单的最友好的模拟器，检查所有的指令错误，不讲究速度。

（3）Sim-bpred

实现一个分支预测分析器。

（4）Sim-cache

这个工具实现 cache 模拟功能，为用户择的 cache 和快表设置生成 cache 统计，其中可能包含两级指令和数据 cache ，还有一级指令和数据快表，不会生成时间信息。

（5）Sim-eio

这个模拟器支持生成外部事件跟踪（EIO traces）和断点文件。外部事件跟踪俘获程序的执行，并且允许被打包到一个单独的文件，以备以后的再次执行。这个模拟器也提供在外部事件跟踪执行中在任意一点做断点。断点文件可被用于在程序运行中启动 simplescalar 模拟器。

（6）Sim-outorder

最完整的工具。支持依序和乱序执行，branch predictor，memory hierarchy，function unit 个数等参数设定。这个模拟器追踪潜在的所有流水（pipeline）操作。

（7）Sim-profile

也叫 functional simulation，但提供较完整的模拟参数，可依照使用者之设定，决定所要模拟之项目，如 instruction classes and addresses、text symbols、memory accesses、branches and data segment symbols 以方便使用者整理收集数据材料。

三　实验内容

（一） SimpleScalar 基本运行模拟

1. 模拟测试程序

我们采用 SimpleScalar 的 sim-cache 模拟器对 spec2000 中的标准测试文件进行测试。我们已经有了写好的运行脚本：run-all,run-gcc,run-gzip 等。所以只需要直接运行这些脚本就能开始正常的模拟运行。这里注意，在压缩包中 ftc.cfg 配置文件中，写的是 sim-order 的配置文件，所以包括了很多的内容，在这次实验中，我们只做 sim-cache 部分的，所以我们重新写一个专用于 sim-cache 的配置文件。当然，在 sim-order 的仿真中，是包括 sim-cache 部分的，所以其实也能够直接用 sim-order 命令来仿真，考虑到把 sim-order 和 sim-cache 分开做，个人觉得意义不大，所以在实验过程中采用了 sim-order 进行仿真。当然如果想分开做的话，我们自己编写的 sim-cache 配置文件包括以下内容：

```
#
#sim-cache configuration
#

#random number generator seed (0 for timer seed)
-seed                           1

#l1 data cache config, i.e..{<config>|none}
-cache:dl1                      dl1:512:64:2:l

#l1 inst cache config, i.e..{<config>|dl1|dl2|none}
-cache:il1                      il1:512:64:2:l

#instruction TLB config, i.e..{<config>|none}
-tlb:itlb                       itlb:16:4096:4:l

#data TLB config, i.e..{<config>|none}
-tlb:dtlb                       dtlb:32:4096:4:l
```

这时，我们必须把可执行文件 run-**中的-config 之后的“ftc.cft”改为“sim-cache.cfg”，即把

```
./simplesim-3.0/sim-outorder -config "ftc.cfg"
```

改为：`./simplesim-3.0/sim-cache -config "sim-cache.cfg"`

而我们只需要不断修改我们的配置文件 sim-cache.cfg 中的相关参数，就能达到我们实验的比较的目的。具体的参数修改配置，下面将详细给出。

2．Cache 配置

在第一次的仿真中，我们没有对 Cache 的结构进行配置，使用了默认的 Cache 配置。在我们的实验中，需要对 Cache 参数进行详细配置。

一般来说，Cache 的结构参数主要包括以下几个方面：容量、块大小、相联度、替换算法等。在 SimpleScalar 模拟器中，采用了两级 Cache 结构，同时数据和指令 Cache 分开。SimpleScalar 的 Cache 参数配置命令为：

<name>:<nsets>:<bsize>:<assoc>:<repl>

其中：

<name> ： Cache的名称，其中:

dl1：一级数据Cache

dl2：二级数据Cache

il1：一级指令Cache

il2：二级指令Cache

dtlb：数据TLB

itlb：指令TLB

<nsets> ： 组的数目

<bsize>：块大小

<assoc> -：相联度

<repl> -：替换策略

此时，Cache容量为：<nsets>*<bsize>*<assoc>

替换策略主要有以下几种：

l ：LRU，最近最少使用

f ： FIFO，先进先出

r ： RANDOM，随机策略

例如:

-cache:dl1 dl1:1024:32:2:l

表示对一级数据 cache 进行配置，1024 表示有 1024 组，Cache 块大小为 32 个

byte，每个组有 2 个 Cache 块，即相联度为 2，替换策略为 LRU。在此配置下，一级数据 Cache 的容量为 1024*32*2=64K byte。

例如对上述矩阵乘测试程序，使用该配置进行模拟执行的命令为：

[root@localhost test]# sim-cache -cache:dl1 dl1:1024:32:2:l a.out

（二） SimpleScalar 模拟统计信息详解

采用上述命令执行后，得到如下所示的信息，其详解如下：

```
[root@localhost test]# sim-cache -cache:dl1 dl1:1024:32:2:l a.out
sim-cache: SimpleScalar/PISA Tool Set version 3.0 of August, 2003.
Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
All Rights Reserved. This version of SimpleScalar is licensed for academic
non-commercial use.    No portion of this work may be used by any commercial
entity, or for any commercial purpose, without the prior written permission
of SimpleScalar, LLC (info@simplescalar.com).

sim: command line: sim-cache -cache:dl1 dl1:1024:32:2:l a.out

sim: simulation started @ Sun Nov 28 22:49:41 2010, options follow:

sim-cache: This simulator implements a functional cache simulator.    Cache
statistics are generated for a user-selected cache and TLB configuration,
which may include up to two levels of instruction and data cache (with any
levels unified), and one level of instruction and data TLBs.    No timing
information is generated.

# -config                                  # load configuration from a file
# -dumpconfig                              # dump configuration to a file
# -h                                 false # print help message
# -v                                 false # verbose operation
# -d                                 false # enable debug message
# -i                                 false # start in Dlite debugger
-seed                                    1 # random number generator seed (0 for timer seed)
# -q                                 false # initialize and terminate immediately
# -chkpt                           <null> # restore EIO trace execution from <fname>
# -redir:sim                       <null> # redirect simulator output to file (non-interactive only)
# -redir:prog                      <null> # redirect simulated program output to file
-nice                                    0 # simulator scheduling priority
-max:inst                                0 # maximum number of inst's to execute
-cache:dl1            dl1:1024:32:2:l # l1 data cache config, i.e., {<config>|none}
-cache:dl2             ul2:1024:64:4:l # l2 data cache config, i.e., {<config>|none}
-cache:il1            il1:256:32:1:l # l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il2                       dl2 # l2 instruction cache config, i.e., {<config>|dl2|none}
-tlb:itlb            itlb:16:4096:4:l # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb            dtlb:32:4096:4:l # data TLB config, i.e., {<config>|none}
-flush                             false # flush caches on system calls
-cache:icompress                   false # convert 64-bit inst addresses to 32-bit inst equivalents
# -pcstat                        <null> # profile stat(s) against text addr's (mult uses ok)

    The cache config parameter <config> has the following format:

        <name>:<nsets>:<bsize>:<assoc>:<repl>

        <name>      - name of the cache being defined
        <nsets>     - number of sets in the cache
        <bsize>     - block size of the cache
        <assoc>     - associativity of the cache
        <repl>      - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random

        Examples:     -cache:dl1 dl1:4096:32:1:l
                      -dtlb dtlb:128:4096:32:r

    Cache levels can be unified by pointing a level of the instruction cache
    hierarchy at the data cache hiearchy using the "dl1" and "dl2"
    configuration arguments.    Most sensible combinations are supported, e.g.,

        A unified l2 cache (il2 is pointed at dl2):
            -cache:il1 il1:128:64:1:l -cache:il2 dl2
            -cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

        Or, a fully unified cache hierarchy (il1 pointed at dl1):
            -cache:il1 dl1
            -cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l


sim: ** simulation statistics **
sim_num_insn                           105663 # total number of instructions executed    总模拟指令条数
```

执行模拟的命令

版权声明等，不用关

sim-cache 的运行命令参数帮助说明，有必要可以

Sim-cache 的 cache 参数配置帮助说明，建议了解。

此处正式开始模拟

模拟总体信息统

```
sim_num_refs                34213 # total number of loads and stores executed    所有的访存指令条数
sim_elapsed_time              200 # total simulation time in seconds    总的模拟时钟周期数
sim_inst_rate             528.3150 # simulation speed (in insts/sec)    模拟速度
il1.accesses               105663 # total number of accesses    一级指令 cache 的访问数目
il1.hits                    90939 # total number of hits    一级指令 cache 的命中数目
il1.misses                  14724 # total number of misses    一级指令 cache 的实效数目
il1.replacements            14468 # total number of replacements    一级指令 cache 上发生替换的数目
il1.writebacks                  0 # total number of writebacks    一级指令 cache 上发生写回的数目
il1.invalidations               0 # total number of invalidations    一级指令 cache 上访存无效的数目
il1.miss_rate              0.1393 # miss rate (i.e., misses/ref)    一级指令 cache 上的失效率
il1.repl_rate              0.1369 # replacement rate (i.e., repls/ref)    一级指令 cache 上的替换概率
il1.wb_rate                0.0000 # writeback rate (i.e., wrbks/ref)    一级指令 cache 上的写回概率
il1.inv_rate               0.0000 # invalidation rate (i.e., invs/ref)    一级指令 cache 上的无效概率
dl1.accesses                34648 # total number of accesses    一级数据 cache 上的总访问数目
dl1.hits                    33972 # total number of hits    一级数据 cache 上的命中次数
dl1.misses                    676 # total number of misses    一级数据 cache 上的实效次数
dl1.replacements                1 # total number of replacements    一级数据 cache 上发生替换的次数
dl1.writebacks                  0 # total number of writebacks    一级数据 cache 上发生写回的次数
dl1.invalidations               0 # total number of invalidations    一级数据 cache 上无效访问的次数
dl1.miss_rate              0.0195 # miss rate (i.e., misses/ref)    一级数据 cache 上的失效率
dl1.repl_rate              0.0000 # replacement rate (i.e., repls/ref)    一级数据 cache 上发生替换的概率
dl1.wb_rate                0.0000 # writeback rate (i.e., wrbks/ref)    一级数据 cache 上发生写回的概率
dl1.inv_rate               0.0000 # invalidation rate (i.e., invs/ref)    一级数据 cache 上发生无效访问的次数
ul2.accesses                15400 # total number of accesses    联合二级 cache 上总的访问次数
ul2.hits                    14280 # total number of hits    联合二级 cache 上命中的次数
ul2.misses                   1120 # total number of misses    联合二级 cache 上失效的次数
ul2.replacements                0 # total number of replacements    联合二级 cache 上发生替换的次数
ul2.writebacks                  0 # total number of writebacks    联合二级 cache 上发生写回的系数
ul2.invalidations               0 # total number of invalidations    联合二级 cache 上无效访问的次数
ul2.miss_rate              0.0727 # miss rate (i.e., misses/ref)    联合二级 cache 上的失效率
ul2.repl_rate              0.0000 # replacement rate (i.e., repls/ref)    联合二级 cache 上发生替换的概率
ul2.wb_rate                0.0000 # writeback rate (i.e., wrbks/ref)    联合二级 cache 上发生写回的概率
ul2.inv_rate               0.0000 # invalidation rate (i.e., invs/ref)    发生无效访问的概率
itlb.accesses              105663 # total number of accesses    指令 TLB 上访问次数
itlb.hits                  105640 # total number of hits    指令 TLB 上命中次数
itlb.misses                    23 # total number of misses    指令 TLB 上失效的次数
itlb.replacements               0 # total number of replacements    指令 TLB 上发生替换的次数
itlb.writebacks                 0 # total number of writebacks    指令 TLB 上发生写回的次数
itlb.invalidations              0 # total number of invalidations    指令 TLB 上发生无效访问的次数
itlb.miss_rate             0.0002 # miss rate (i.e., misses/ref)    指令 TLB 上的失效率
itlb.repl_rate             0.0000 # replacement rate (i.e., repls/ref)    指令 TLB 上的替换概率
itlb.wb_rate               0.0000 # writeback rate (i.e., wrbks/ref)    指令 TLB 上写回的概率
itlb.inv_rate              0.0000 # invalidation rate (i.e., invs/ref)    指令 TLB 上无效访问的概率
dtlb.accesses               34648 # total number of accesses    指令 TLB 上总的访问次数
dtlb.hits                   34636 # total number of hits    数据 TLB 上总的访问次数
dtlb.misses                    12 # total number of misses    数据 TLB 上总的失效次数
dtlb.replacements               0 # total number of replacements    数据 TLB 上总的替换次数
dtlb.writebacks                 0 # total number of writebacks    数据 TLB 上发生写回的次数
dtlb.invalidations              0 # total number of invalidations    数据 TLB 上发生无效访问的次数
dtlb.miss_rate             0.0003 # miss rate (i.e., misses/ref)    数据 TLB 上失效率
dtlb.repl_rate             0.0000 # replacement rate (i.e., repls/ref)    数据 TLB 上发生替换的概率
dtlb.wb_rate               0.0000 # writeback rate (i.e., wrbks/ref)    数据 TLB 上发生写回的概率
dtlb.inv_rate              0.0000 # invalidation rate (i.e., invs/ref)    数据 TLB 上发生无效访问的概率
ld_text_base            0x00400000 # program text (code) segment base
ld_text_size                96544 # program text (code) size in bytes
ld_data_base            0x10000000 # program initialized data segment base
ld_data_size                12288 # program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base           0x7fffc000 # program stack segment base (highest address in stack)
ld_stack_size               16384 # program initial stack size
ld_prog_entry           0x00400140 # program entry point (initial PC)
ld_environ_base         0x7fff8000 # program environment base address address
ld_target_big_endian            0 # target executable endian-ness, non-zero if big endian
mem.page_count                 36 # total number of pages allocated
mem.page_mem                  144k # total size of memory pages allocated
mem.ptab_misses                37 # total first level page table misses
mem.ptab_accesses        1092117 # total page table accesses
mem.ptab_miss_rate         0.0000 # first level page table miss rate

[root@localhost test]#
```

一级指令 cache 的性能统计信息

一级数据 cache 的性能统计信息，重点关

二级 cache 的性能统计信息

指令 TLB 的性能统计

数据 TLB 的性能统计信息

其他内存和基地址等信息统计

（三） Cache 容量对性能的影响

1．操作方法与实验步骤

（1）改变 SimpleScalar 模拟器的一级数据 cache dl1 的容量配置，在 SimpleScalar

的配置过程中，固定 cache 块大小、相联度、替换策略等参数，通过改变组数来改变 Cache 的容量大小，执行程序，具体运行配置参数，只需要在 ftc.cfg 中的

```
# l1 data cache config, i.e., {<config>|none}
-cache:dl1              dl1:512:64:2:l
```

后面 4 组参数，同时，我们现在只是测试只有一级 cache 的情况，所以我们把 ftc.cfg 中的有关二级 cache 的相关参数都注释掉。如下所示：

```
# l2 data cache config, i.e., {<config>|none}
#-cache:dl2              ul2:2048:64:8:l

# l2 data cache hit latency (in cycles)
#-cache:dl2lat                      13

# l2 data cache write back redundancy size (in blocks)
#-cache:dl2nwb                      32

# l2 data cache bch size (in blocks)
#-cache:dl2nbch                     32

# l2 data cache bch decoding latency (in cycles)
#-cache:dl2bchlat                   72
```

2.实验参数修改过程：

在确定块大小为 32byte,相连度为 2，替换策略为 LRU 情况之下，把参数改为如下所示：

块大小=32byte,相连度=2，替换策略=LRU	
dl1 容量	修改命令
4k	dl1:64:32:2:1
8k	dl1:128:32:2:1
16k	dl1:256:32:2:1
32k	dl1:512:32:2:1
64k	dl1:1024:32:2:1
128k	dl1:2048:32:2:1

3.结果分析

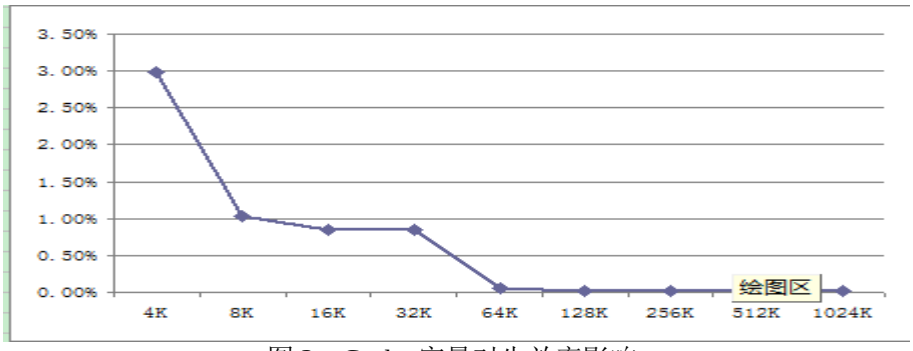


图 3　Cache 容量对失效率影响

观察 Cache 容量对性能的影响，将测得的失效率绘制为曲线，得到如上图所示

的曲线图。从图中可以看出，随着 Cache 的容量不断增加，程序的失效率不断降低。由此可见，Cache 的容量对于 Cache 性能有着重要影响。一般来说，容量越大，Cache 性能越好，发生失效的概率就越低。但对于某一个应用来说，容量达到一定程度后，性能改善程度会达到峰值。但在容量达到 128K byte 后，失效率就不再降低，稳定在一个固定范围内，说明此时容量已经不是 Cache 性能的瓶颈了，增大 Cache 容量对性能提升已无促进作用。

（四） Cache 相联度对性能的影响

1．操作方法与实验步骤

（1）改变 SimpleScalar 模拟器的一级数据 cache dl1 的容量配置，在 SimpleScalar 的配置过程中，固定容量大小、cache 块大小、替换策略等参数，改变相联度。

2.实验参数修改过程：

容量=16k,块大小=32byte,替换策略=LRU	
相关度	修改命令
1	dl1:512:32:1:1
2	dl1:256:32:2:1
4	dl1:128:32:4:1
8	dl1:64:32:8:1
16	dl1:32:32:16:1
32	dl1:16:32:32:1
64	dl1:8:32:64:1
128	dl1:4:32:128:1
256	dl1:2:32:256:1

3.结果分析

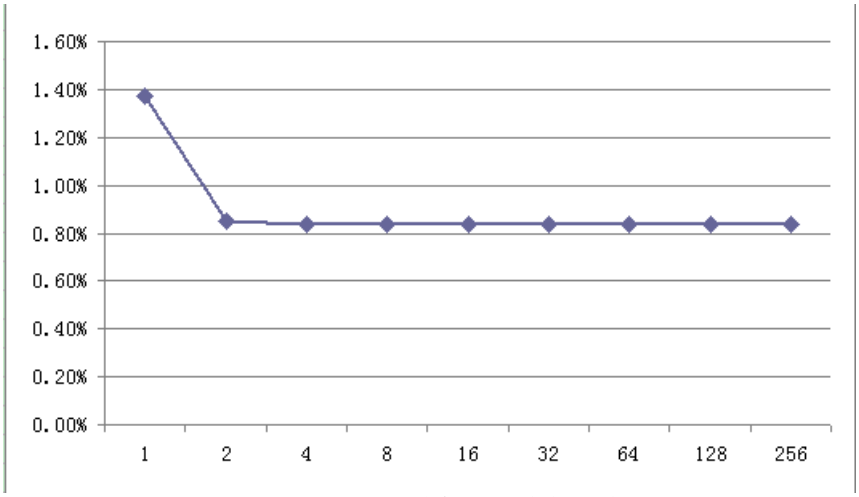


图 4　Cache 相联度对失效率影响

在组相联 Cache 中，相联度对 Cache 性能也有着重要影响，因为相联度直接决定 Cache 中发生冲突失效的概率。相联度越高，块冲突概率就越低，因而 Cache 的失效率就越低。块冲突是指一个主存块要进入已被占用的 Cache 块位置。显然，全相联(相联度为 n)的失效率最低，直接映像(相联度为 1)失效率最高。虽然从降低失效率的角度来看，相联度越大越好，但从我们的实验将可以发现，增大相联度并一定使 Cache 的性能提高，反而会使系统复杂性增加，这点将从我们的实验结果中得到具体反映。由观察得，具体为在 16K byte 容量下，将相联度从 1 变化到 256。由

于容量一定，组数也从 512 变化到 2。在不同 Cache 相联度的配置下，其失效率变出，如表 2 所示，将这些失效率绘制为曲线，得到图 4 所示的曲线。根据这些实验结果我们可以发现，在相联度 4 以下，失效率会随着相联度的增加而降低。而相联度超过 4 以后，相联度的增加并不会使失效率继续降低，反而稳定在一个固定水平上。从上述实验结果可以发现，相联度的增加在一定的范围内可以促进 Cache 性能的改善，但这个范围非常有限，仅在 1 路、2 路或 4 路组相联上起作用。

（五） Cache 块大小对性能的影响

1．操作方法与实验步骤

（1）改变 SimpleScalar 模拟器的一级数据 cache dl1 的容量配置，在 SimpleScalar 的配置过程中，相联度、替换策略等参数，对于 2k、4k、8k 一直到 128k，分别改变 cache 块大小，执行程序。

2.实验参数修改过程：

容量=2k,关联度=2,替换策略=LRU	
块大小	修改命令
8byte	dl1:128:8:2:1
16byte	dl1:64:16:2:1
32byte	dl1:32:32:2:1
64byte	dl1:16:64:2:1
容量=4k,关联度=2,替换策略=LRU	
块大小	修改命令
8byte	dl1:256:8:2:1
16byte	dl1:128:16:2:1
32byte	dl1:64:32:2:1
64byte	dl1:32:64:2:1
容量=8k,关联度=2,替换策略=LRU	
块大小	修改命令
8byte	dl1:512:8:2:1
16byte	dl1:256:16:2:1
32byte	dl1:128:32:2:1
64byte	dl1:64:64:2:1
容量=16k,关联度=2,替换策略=LRU	
块大小	修改命令
8byte	dl1:1024:8:2:1
16byte	dl1:512:16:2:1
32byte	dl1:256:32:2:1
64byte	dl1:128:64:2:1
容量=32k,关联度=2,替换策略=LRU	
块大小	修改命令
8byte	dl1:2048:8:2:1
16byte	dl1:1024:16:2:1
32byte	dl1:512:32:2:1
64byte	dl1:256:64:2:1
容量=64k,关联度=2,替换策略=LRU	
块大小	修改命令

8byte	dl1:4096:8:2:1
16byte	dl1:2048:16:2:1
32byte	dl1:1024:32:2:1
64byte	dl1:512:64:2:1
容量=128k,关联度=2,替换策略=LRU	
块大小	修改命令
8byte	dl1:8192:8:2:1
16byte	dl1:4096:16:2:1
32byte	dl1:2048:32:2:1
64byte	dl1:1024:64:2:1

3.结果分析

降低 Cache 失效率最为简单的办法就是增加块大小。具体实验方法为测试在不同容量情况下，固定容量大小、替换策略、相联度等，改变 Cache 的块大小，观察 Cache 的失效率变化。

经过测试，我们得到各个容量下失效率随块大小的变化情况。具体如图 5 所示，每条不同颜色的曲线代表在某一 Cache 容量下，失效率随块大小的变化情况。从图中可以发现两点规律：

(1) 对于给定的 Cache 容量，当块大小增加时，失效率开始时处于下降趋势，后来反而会上升。

(2) Cache 容量越大，会使失效率达到最低的拐点的块大小增大。

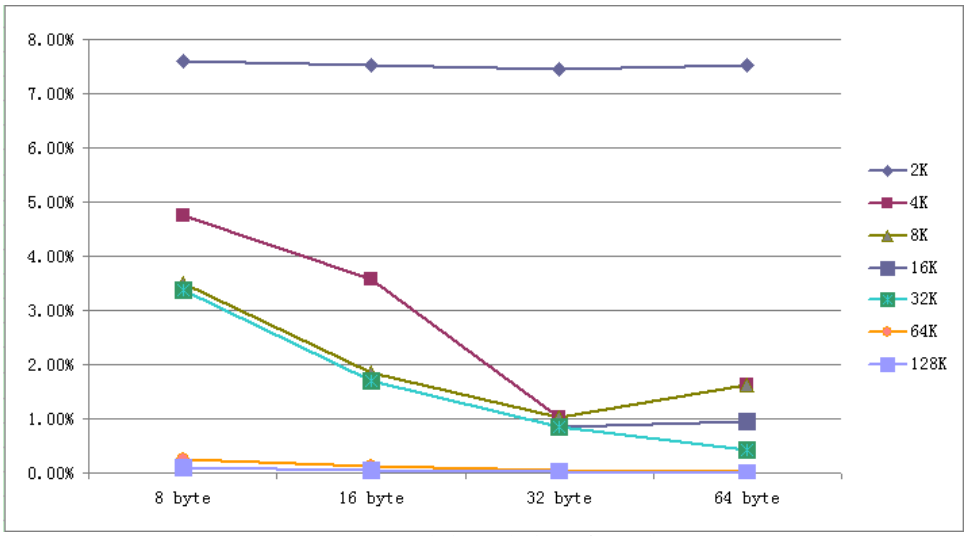


图 5 失效率随块大小变化情况

导致上述失效率先下降后上升的原因在于增加块大小会产生双重作用。一方面会减少强制性失效，因为程序局部性原理，增加块大小增加了利用空间局部性的机会；另一方面，在容量一定情况下，增加块大小会减少总的块数目，会增加冲突失效，在 Cache 容量较小时，还可能增加容量失效。刚开始增加块大小时，由于块大小还不是很大，上述第一种作用超过第二种作用，使失效率降低。当块大小增加到一定程度时，第二种作用会超过第一种作用，使失效率上升。

（六）Cache 的替换策略对性能的影响

1．操作方法与实验步骤

（1）改变 SimpleScalar 模拟器的一级数据 cache dl1 的容量配置，在 SimpleScalar 的配置过程中，将 Cache 的块大小固定为 32byte，相联度固定为 2，对于 2k、4k、8k 一直到 256k，分别改变替换策略，执行程序。

2.实验参数修改过程：

容量=2k,块大小=32byte,关联度=2	
替换策略	修改命令
RANDOM	dl1:32:32:2:r
FIFO	dl1:32:32:2:f
LRU	dl1:32:32:2:1
容量=4k,块大小=32byte,关联度=2	
替换策略	修改命令
RANDOM	dl1:64:32:2:r
FIFO	dl1:64:32:2:f
LRU	dl1:64:32:2:1
容量=8k,块大小=32byte,关联度=2	
替换策略	修改命令
RANDOM	dl1:128:32:2:r
FIFO	dl1:128:32:2:f
LRU	dl1:128:32:2:1
容量=16k,块大小=32byte,关联度=2	
替换策略	修改命令
RANDOM	dl1:256:32:2:r
FIFO	dl1:256:32:2:f
LRU	dl1:256:32:2:1
容量=32k,块大小=32byte,关联度=2	
替换策略	修改命令
RANDOM	dl1:512:32:2:r
FIFO	dl1:512:32:2:f
LRU	dl1:512:32:2:1
容量=64k,块大小=32byte,关联度=2	
替换策略	修改命令
RANDOM	dl1:1024:32:2:r
FIFO	dl1:1024:32:2:f
LRU	dl1:1024:32:2:1
容量=128k,块大小=32byte,关联度=2	
替换策略	修改命令
RANDOM	dl1:2048:32:2:r
FIFO	dl1:2048:32:2:f
LRU	dl1:2048:32:2:1
容量=256k,块大小=32byte,关联度=2	
替换策略	修改命令

RANDOM	dl1:4096:32:2:r
FIFO	dl1:4096:32:2:f
LRU	dl1:4096:32:2:1

3.结果分析

由于主存中的块比 Cache 中的要多，所以当要从主存调入一个块到 Cache 中时，会出现该快所映像到的一组 Cache 块已被占用的情况。这是需要强制其中的一块移出 Cache，以接纳新的 Cache 块。这就需要替换策略选择替换的块。

替换策略主要是在对 Cache 块进行淘汰时，如何选择要替换的块的策略。目前主要又三种替换策略：最近最少使用(LRU)策略、先进先出(FIFO)策略、随机(RANDOM)策略。这三种替换策略各有优劣。好的替换策略会将使用率高的 Cache 块更长时间的驻留在 Cache 中，从而降低 Cache 失效率，提高 Cache 性能。为比较上述三种替换策略对 Cache 性能的影响，我们对他们分别进行实验比较。

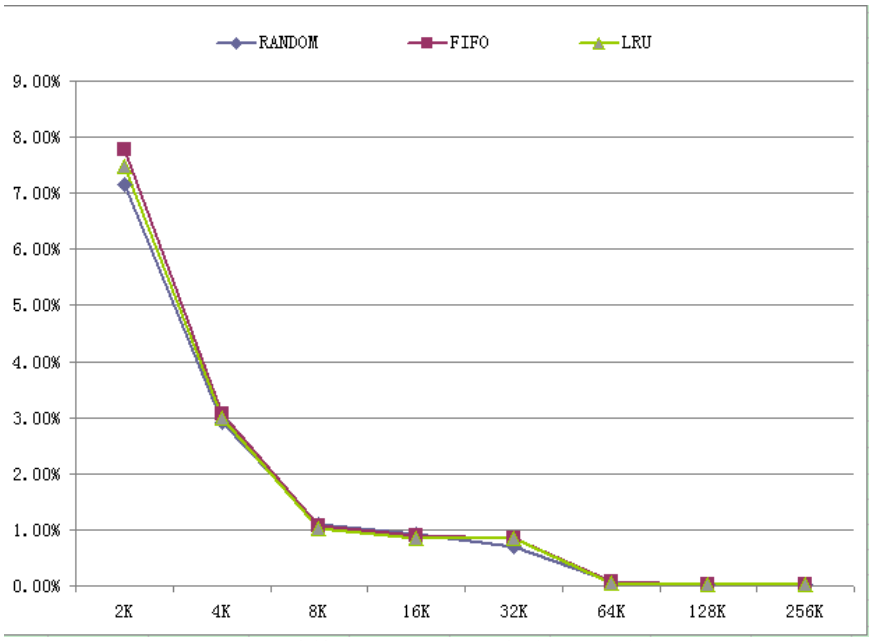


图 6 替换策略对 Cache 性能的影响

从图 6 中可以发现，在 Cache 容量较小的情况下，随机(RANDOM)策略相对较好，而随着 Cache 容量的增加，最近最少使用(LRU)和先进先出(FIFO)策略的效果较好。所以一般的计算机较多的采用 LRU 替换策略。

（七）二级 Cache 对性能的影响

1. 操作方法与实验步骤

以上部分都是我们只考虑一级 cache 的仿真结果，接下来，我们考虑分级 cache 对性能的影响。Cache 分级结构的主要优势在于，对于一个典型的一级缓存系统的 80％的内存申请都发生在 CPU 内部，只有 20％的内存申请是与外部内存打交道。而这 20％的外部内存申请中的 80％又与二级缓存打交道。因此，只有 4％的内存申请定向到 DRAM 中。Cache 分级结构的不足在于高速缓存组数目受限，需要占用线

路板空间和一些支持逻辑电路，会使成本增加。

我们把之前在 ftc.cfg 里面注释掉的关于二级 cache 的全部修改回来，两级 cache 的参数是完全相同的，所以我们只需要重复以上的步骤，就能看出两者之间不同，具体过程看以上部分，不再写出。

（八）sim-outorder 的仿真

Sim-outordersim-outorder 是 simplescalar 中最完整的模拟器，支持依序和乱序执行，branch predictor，memory hierarchy，function unit 个数等参数设定。这个模拟器追踪潜在的所有流水（pipeline）操作。要实现对指令的乱序执行，sim-outorder 中需要设置很多不同的功能单元，五种很重要的功能单元支持 sim_outorder 对指令序列的乱序执行：保留站与重定序缓冲（RUU)、Load/Store 队列（LSQ)、取指队列、输入输出相关链和寄存器忙闲表。它们在 simplescalar 中是通过五种数据结构来实现的。

RUU（Register update unit）单元实现寄存器的同步和通讯功能，它将再定序缓冲和保留站统一起来，作为一个循环队列来管理。RUU 队列记录了指令的操作类型、源操作数、数据效性标识。其中的数据项在指令发射时分配，在提交时回收；当寄存器数据和存储器数据相关性满足时，实现乱序流出；Load/Store 队列（LSQ）处理存储器的相关性问题。如果 store 操作是猜测执行的，其值就被放入队列中。当之前的写入地址都已知之后，Load 操作就可以访存。如果地址匹配，load 操作可以在存储系统或者 Load/Store 队列中以前的 store 值的允许下进行；取指队列是由取指段建立，在调度段译码并调度的指令队列；没被调度的指令仍留在其中。它是用一个结构数组来实现的；输入输出相关链，是用来记录前一条指令的输出数据（结果操作数与后几条指令的输入数据）源操作数的相关性的链表；寄存器忙闲表，是用来记录当前各个寄存器被哪一条指令占用的结构数组。

因为功能强大，所以它的可配参数就会很多，有兴趣的同学，再详细的对它理解。