Address-Trace Based Cache Profiling

Maz Spork (halgrim@diku.dk)

March 19, 1999

Abstract

Given a set of constants describing the configuration of a cache, as well as a list of read/write-augmented memory references, a *cache profiling tool* is able to generate the number and types of cache misses for a given program. This note presents such a tool for use in C++ programs.

1 Basics

A cache is statically configured from the following constants (*cache metrics*):

M, the total capacity in words;

B, the size of each block, or line, in the cache, which is the word quantity read from or written to memory in a *transaction*;

a, the associativity of the cache; and

p, the replacement policy.

The number of sets in the cache is now $s = M/(aB)$, and the total capacity of the cache, in bytes, is $bBM/8$, given that b is the word size in bits. The replacement policy is typically chosen as either *random* (RND), *not recently used* (NRU), or *least recently used* (LRU). This is a fixed replacement policy and is not under software control.

A reference pattern $R = \{r_1, r_1, \ldots, r_N\}$ is also given, where each reference $r = (A, t, w)$ such that A_i is the referential (virtual) address, t_i is a time-stamp, and w_i is a boolean false for a read access and true for a write. R is ordeded on t.

A cache block c consists of a tag field $\mathrm{TAG}(c)$, a valid bit $\mathrm{VAL}(c)$, and a dirty bit $\mathrm{DRT}(c)$ as well as B words of data, $D(c) = d_0(c)$

through $d_{B-1}(c)$. TAG(c) identifies the address of $d_0(c)$ in (virtual) memory, VAL(c) is true iff $D(c)$ contains valid data, and DRT(c) is true iff any element of $D(c)$ has been modified since it was read from main memory. In addition, c may contain a time stamp REF(c) with information about the time of the last reference (either read or write) to $D(c)$. REF(c) is not present if the $p = $ RND, it is a single bit if $p =$ NRU, and may be a wider time stamp if $p =$ LRU.

2 Addresses

Each memory reference A is split into components, which are adjacent bit fields, $A = A_{tag}||A_{idx}||A_{off}$, where $||$ means concatenation. The width in bits of these fields are given by the cache metrics:

$$
\begin{aligned}
|A_{off}| &= \log_2 B \\
|A_{idx}| &= \log_2 s \\
|A_{tag}| &= b - |A_{off}| - |A_{idx}|
\end{aligned}
$$

and their values are computed according to the cache metric thus:

$$
\begin{aligned}
A_{off} &= A \textbf{ mod } B \\
A_{idx} &= (A \textbf{ div } B) \textbf{ mod } s \\
A_{tag} &= A \textbf{ div } (Bs)
\end{aligned}
$$

tag	*idx*	*off*

Figure 1: Bit fields of a virtual address.

3 Hits and misses

For a given reference r_i, its corresponding cache set $S = \{c_0, \ldots, c_{a-1}\}$ is given by the index A_{idx}. The reference represents a *cache hit* iff there exists a tag value $t \in \{\text{TAG}(c) \mid c \in S\}$ such that $t = \text{TAG}(A_i)$ and VAL(t) is **true**. Otherwise, the reference is a *cache miss*.

3.1 Hits

A cache hit is free.

3.2 Misses

The consequence of a cache miss is (a) the *selection* of a new cache block; (b) reading the contents of the block from memory; (c) writing the previous contents of the block to memory if necessary[1]; and (d) updating the fields of the block.

Selection of the cache block $s \in S$ within that set depends on the replacement policy p:

RND: the block is chosen randomly between 0 and $a - 1$.

NRU: the block is chosen randomly between the blocks $S \backslash c_{last}$, where c_{last} is the cache block last referenced within S.

LRU: the block is chosen as the block in S with the smallest $\mathrm{REF}(c_i)$, that is c_i where $\mathrm{REF}(c_i) = \mathbf{min}\ \{\mathrm{REF}(c_0), \ldots, \mathrm{REF}(c_{a-1})\}$.

The offset within the cache line is A_{off}. Memory is byte-addressed.

4 Profiling C++ programs

The cache profiling is a two-stage task: Collecting the reference pattern from a trial run of a program; and statically analyzing this reference pattern.

4.1 Recording the reference patterns

The access pattern R is recorded by means of calling special functions for each read and write found in the include file `cprof.h`. These function calls are inserted manually by the programmer:

`READ(x)` read the variable `x`.

`WRITE(x,v)` write the value `v` to the program variable `x`.

A symbol `CPROF` must be defined if an address trace is desired. The definition of CPROF is the filename to which the address trace should be written, in double quotes. For example,

```
void main () {
  const unsigned n = 32;
  int i, v[n];
  for (i = 1; i < n; i++)
    v[i-1] = v[i];
}
```

[1]The order of the read and write transaction depends on the memory system's ability to *queue* the write transactions in a special write buffer in order to give priority to reads. Depending on this ability, the miss penalty for a *dirty miss* can be assumed equal to that of a *clean miss*.

becomes

```
#define CPROF "mytrace.dat"
#include "cprof.h"
void main () {
  const unsigned n = 32;
  int i, v[n];
  for (WRITE(i,1); READ(i) < n; WRITE(i,READ(i)+1))
    WRITE(v[READ(i)-1],READ(v[READ(i)]));
}
```

If `CPROF` is not defined, the function calls impose no overhead on the program's execution time. If the symbol is defined, the trace file (`mytrace.dat` in this example) will contain something like

```
r 0xbffff928
r 0xbffff928
r 0xbffff930
r 0xbffff928
w 0xbffff92c
r 0xbffff928
w 0xbffff928
r 0xbffff928
r 0xbffff928
...
```

It may not be necessary, however, to insert `READ` and `WRITE` statements for every lvalue and rvalue in the program. In particular, data allocated on the stack, such as the control variable `i` in the example, are typically register allocated and can be assumed transparent in cost in respect to the cache. In the example, it is the array `v[]` that is the focal point and the program might be rewritten as

```
#define CPROF "mytrace.dat"
#include "cprof.h"
void main () {
  const unsigned n = 32;
  int i, v[n];
  for (int i = 1; i < n; i++)
    WRITE(v[i-1],READ(v[i]));
}
```

The choice of which memory accesses to include is left to the programmer.

The output string can be converted to GNUPLOT input using a supplementary tool *gpconvert* to facilitate plotting the reference patterns.

4.2 Profiling

The reference pattern is fed to the static analyzer `cprof` which will collect statistics for a given cache configuration. `cprof` takes the following optional command-line arguments:

`-M`*M*	set cache capacity to M words.
`-a`*a*	set associativity to a.
`-b`*B*	set block size to B words.
`-p`*p*	set replacement policy (`rnd`, `nru` or `lru`).
`-w`*n*	set word size to n bits.
`-f`*file*	set the input file to *file*.
`-h`	help
`-v`	verbose output (all memory references).

If associativity is set to 1, the cache will be direct-mapped; if it is set to M/B (or 0, for convenience), it will be fully associative. All other values produces an a-way set-associative cache.

The profiler reports the total number of references; the hit and miss count; and the types of misses (compulsory, capacity and conflict), all for both read and write accesses. For example,

```
$ cprof -fextrace.dat -a4 -plru -m8192
Cache profile: extrace.dat
--------------------------
Cache metrics:
- capacity:         8192 words (32 kb)
- associativity:    4-way
- bits per word:    32
- bytes per block:  32
- words per block:  8
- cache sets:       256
- replacement:      lru

Statistics:
- references:   316972 ( 245777 reads,  71195 writes )
- hit count:    262751 ( 201437 reads,  61314 writes )
- miss count:
  - compulsory    1024 (   1024 reads,      0 writes )
  - capacity     51530 (  41651 reads,   9879 writes )
  - conflict      1667 (   1665 reads,      2 writes )
  - total        54221 (  44340 reads,   9881 writes )
- hit/miss:   4.84593
```

4.3 Limitations

In order for the address trace to be faithful it is further necessary to insert the statements carefully with respect to the data types in question. An assignment statement `a = b`, where `a` and `b` are multi-word structures, will produce a read sequence for all words in `b` followed by a write sequence for all words in `b` (the word size is assumed to be `sizeof(void*)`). This is not an accurate sequence, however, because a block copy is an intermixed sequence of reads and writes.

5 Availability

The source code is available at the Worldwide Web and is standard C++ source: `http://www.diku.dk/students/halgrim/cprof.tar`.