

Data Cache Optimization

NDK 5.7/MPTK 2.4

Application Note, 12 December 2007

Overview

Optimising the data cache usage of a program running on a Nexperia Media Processor has always been an advanced topic. This app note, along with a matching set of examples, is designed to provide you with a framework that can be used in your own development. Using a simple example, it shows how to setup and use the tools.

The code is provided in three versions. One of these is designed to run on the hardware and in a multi-tasking environment with an NDK makefile. A second is designed to run with the latest tools on the simulator. The third version uses older tools and can be helpful in case you need to migrate from the older tools.

This app note does not cover how the data cache works. While the examples presented here include some interesting details, this app note assumes you are familiar with the details of the cache. These are described in the hardware databooks for the cores in question, and in the .ppt training slides provided with NDK. The training class provided by Chris Bore probably gives you your best chance to really discuss and understand the details of the cache's operation. This app note is focused on how to use the tools.

As is typical for optimization problems, we start simple and one step at a time, we make things more complex. At each step, we consider what we expect, and then we look for evidence of that in the simulation results. Sometimes what we find is different from what we expect. This tells us that we must refine our assumptions.

1.1 PACKAGE CONTENTS

The package provided here includes an example application and two new components. They are designed to be used with NDK 5.5, though they are easily adapted to earlier NDK versions. As always, you are encouraged and expected to have access to the latest release for reference.

1 Apps/exHwPrefetch/verilator

This is the legacy version. It uses the old verilated simulator. This might be the simplest version of the demo. It is the legacy version. The verilated simulator uses the verilog model of the CPU core. This makes it reasonably accurate when trying to understand the behavior of the cache. It does not support conveniences like file IO, and this can make it hard to use with more complex algorithms.

2 Apps/exHwPrefetch/verisim

This version is updated to run on tmXxxVerisim. It is very similar to the previous verilator version. The difference is that it uses normal TCS simulators. As of TCS 5.01, all of the features of the verilated simulators are incorporated in the main TCS simulators. This allows you to trade off speed for accuracy, and it gives you access to conveniences such as file IO.

3 Apps/exHwPrefetch

This is the NDK version of the example. It demonstrates how to use the HW prefetch features and the measuring tools on the HW in the NDK environment. While it might be modified to run on the simulator, its main purpose is to run on the hardware.

4 Comps/tmdPrefetchRegion

A component that allows you to assign different prefetch regions to different tasks in an multitasking environment

5 Comps/tmTaskSwitch

A component used by tmdIPrefetch to install a function that is called on every task switch. This component allows multiple clients to install functions to be called when tasks switch.

2.0 ABOUT THE ALGORITHM

The core algorithm demonstrated in this example computes the average of two input frames and stores it in an output frame. The actual meat of the algorithm is the mix() function. Processing is done in 8x8 pixel blocks (8 bit per pixel), from top left to bottom right. This is what we call the "walking pattern". The PNX1700 uses the tm5250 core, and its hardware prefetch facility is designed to aid just such an access pattern. So we use this to demonstrate hardware prefetch.

In each version of the code, you will see three steps demonstrated. Here I reproduce the analysis found with the TCS 4.61 verilator version.

2.1 STEP 1

Let's first run the example with the default hardware prefetch settings, i.e. fetch the next sequential cache lines, and see what happens. Assume we have chosen a stride of 768 bytes and the frames are 720 pixels (bytes) wide and 576 lines high.

Figure 1: Theoretical behavior with default prefetching settings.

The figure above shows what we would expect to see in terms of cache behavior under ideal conditions, i.e. assume a memory latency of 0 cycles (m=0). We expect that the first cache line over the entire height of the frame (blue area) except every 8th line (not depicted) will generate a miss because we are fetching the next sequential cache line. Every 8th line we will also get a hit because we are processing rows of 8 lines high and the last cache line in this row will trigger a hardware prefetch for the first cache line of the next row of 8 lines. The other part of the frame (red area) is expected to be already in the cache when we need it. If we add this up we expect $576 - 576/8 + 1$ misses per frame, i.e. 505 misses per frame, 1010 misses for the two frames together.

Table 1: L2 cache read miss and prefetch summary for step 1 with 0 latency.

L2 Data Cache Read Miss summary:

0	1	2	3
0 1 2 3 4 5 6 7 8 9 A B C D E F	0 1 2 3 4 5 6 7 8 9 A B C D E F	0 1 2 3 4 5 6 7 8 9 A B C D E F	0 1 2 3 4 5 6 7 8 9 A B C D E F
E F 0 1 2			

Now let us run the same example with a more realistic (on the pessimistic side) memory latency number, i.e. 75 cycles. The table below (from mix_step1.c.csv, the description of the _mix function, near line 403) shows the performance numbers for this case. Note that the memory latency is specified to the simulator as a command line parameter.

Table 2: Performance numbers with default prefetch settings. (verilator version)

name	instr	stalls	I\$	D\$	D\$ L2 miss	D\$ L2 hw pf done
Mix	370154	427240	3645	424459	1306	5608

Assuming this function runs at a frame rate of 50 frames per second, we see a total load of roughly 40Mhz, with 21Mhz of D\$ stalls.

2.2 STEP 2

Table 3: Prefetch programming example for step 2.

```
// Program preftcher to fetch 8 lines ahead for both prv and cur frame
#define PF_LINES 8
MMIO(TM_PF0_START_ADDRM(0)) = (int)(prv);
MMIO(TM_PF0_END_ADDRM(0)) = (int)(prv + ((ny - PF_LINES) * i_strd) - 1);
MMIO(TM_PF0_CTLM(0)) = (1 << 31) | (PF_LINES*i_strd);

MMIO(TM_PF1_START_ADDRM(0)) = (int)(cur);
MMIO(TM_PF1_END_ADDRM(0)) = (int)(cur + ((ny - PF_LINES) * i_strd) - 1);
MMIO(TM_PF1_CTLM(0)) = (1 << 31) | (PF_LINES*i_strd);
```

Instead of using the default hardware prefetch settings, we now program the hardware prefetch feature to operate with a stride of 8 video lines (8*6 cache lines). We still assume that we have chosen a stride of 768 bytes and the frames are 720 pixels (bytes) wide and 576 lines high.

The code fragment show above (table 3, found in mix.c) shows how the prefetch regions are being programmed. We use one region per input frame. We leave regions 3 and 4 untouched. Note that we program the end address to 8 lines above the lower right corner of the frame. This is done so that we do not trigger prefetches outside the frame boundary because we do not want to waste memory bandwidth by generating more traffic than necessary.

Figure 2: Theoretical behavior with a prefetching stride of 8 lines.

The figure above shows what we would expect to see in terms of cache behavior under ideal conditions, i.e. again assume a memory latency of 0 cycles. We expect that only the first 8 video lines will yield misses in the cache. This means per frame we expect 8*6 is 48 misses, so 96 misses for two frames.

Table 4: L2 cache read miss and prefetch summary for step 2 with 0 latency.

The table above (mix_step2.c.csv) shows the performance numbers for a memory latency number of 75 cycles. We see that we still have many more misses than our theoretical case, but compared to the default prefetching scheme we cut the number of L2 misses in half. This results in a drop in D\$ stalls from 424Kcycles to 363Kcycles, a 15% performance improvement in D\$ stalls. Assuming again this function runs at a frame rate of 50 frames per second, we now see a total load of roughly 37Mhz, with 18Mhz of D\$ stalls, yielding a 9% overall performance improvement.

2.3 STEP 3

Looking at the performance numbers of step 2 in more detail we see that there are a great deal of copyback buffer full stall cycles (131Kcycles). By looking at what happens when processing the first 8 by 8 pixel block at the beginning of a cache line we can understand what is causing this and take measures to prevent this from happening.

Figure 3: 8 by 8 block on a cache aligned grid.

Because the block is cache aligned (see picture above), processing each block triggers 8 hardware prefetch requests. The FIFO for hardware prefetch requests is only 6 deep so 2 requests are discarded. These requests will not be reposted when processing the next block. Besides triggering 8 hardware prefetches, we might also trigger copybacks of 8 cachelines with dirty data. But there is only a 2 deep FIFO to hold copyback requests. The discarded prefetch and copybacks requests are causing extra stalls.

Figure 4: 8 by 8 block on a non cache aligned grid.

If instead of a cache line aligned frame stride (768 bytes), we use a non-aligned stride (e.g. 720 bytes) as depicted in the picture above, then both the prefetch requests and copybacks will be spaced out in time.

Table 6: Performance numbers with a prefetching stride of 8 lines, frame stride is 720 bytes.

name	Instr	stalls	I\$	D\$	D\$ L1	cb full	D\$ L2 miss	D\$ L2 hw pf done
mix	370158	287338	3755	284659	7690	860	5622	

With the stride changed from 768 to 720 bytes, we now see the copyback buffer full stall cycles drop to around 8K cycles. This results in a drop in D\$ stalls from 363K cycles in step 2 to 285K cycles, a 22% performance improvement in D\$ stalls. Assuming again this function runs at a frame rate of 50 frames per second, we now see a total load of roughly 33Mhz, with 14Mhz of D\$ stalls, yielding a 11% overall performance improvement compared to step 2. Compared to our starting point, we achieved an 18% performance improvement for this function.

3.0 COMPARING TOOL VERSIONS

3.1 USING SIM5250 AND TCS 4.61

TCS 4.61 was the first release to support the tm5250 core in the PNX1700. tm5250verisim was not yet ready when this was released. Hence the earlier sim5250 was used. The tm5250 specific optimisations performed by TCS 5.01 are not that different from TCS 4.61. TCS 5.01 introduced

more optimisations at a higher level, from tmcc, not from tmsched. While the format of a .o file is different and hence you cannot mix .o files from TCS 4.x with those from TCS 5.x, the format of the out files is the same. You can use either simulator with the output of either compiler.

3.2 USING TM5250VERISIM AND TCS 5.01

The verisim directory gives sources and makefiles updated to use tm5250verisim and TCS 5.01. The same three steps are demonstrated.

Using tm5250verisim to exactly simulate the HW can be slow. You must ask yourself what information you want and then set up the simulators to get that information as quickly as possible. For instance, running the three steps of the simulation with

```
tm5250verisim -m 75-fr 500 200 200
```

took nearly an hour on a reasonably modern machine.

```
tm5250verisim -m 75
```

took 20 minutes on that same machine.

The "sim5250" version with TCS 4.61 took 13 minutes on the same machine.

```
tm5250sim -m 75
```

took 2 minutes on that same machine.

The first of these command lines (-fr) should simulate very nearly the actual behavior of the hardware. In fact, it does not using the version of tm5250verisim. The last of these examples does not use the verilog model and hence does not give you any useful information about how the data cache works. The results from the middle example give you enough data to learn what you need to know about the algorithm. As you can see in the following material, they match the hardware pretty well.

The second option, tm5250verisim -m 75 gave these results. You can see that they are very similar to those from the old verilator.

Table 7: Performance numbers with default prefetch settings. (verisim vs verilator version)

Name	instr	stalls	I\$	D\$	D\$ L2 miss	D\$ L2 hw pf	done
Mix (verlator)	37015442724036454244591306	5608					
Mix (verisim)	37015443052337814276221300	5612					

The memory trace file generated for step 1 (mix_step1_mem.txt) also looks very similar to that presented in table 1.

The results for step 2 are also similar in nature. It is interesting to note that there are differences. First, there are differences between the results shown above in table 4, which are perfect, and the results actually seen, which are similar, but not perfect. There are also differences between the result found with 4.61 and verilator and the results found with 5.01 and verisim. The updated compiler may very well cause some of these, though. To understand this difference, I bring up the two .csv traces of the mix function. That generated by verilator, and that generated by verisim.

First, I can see by scrolling to the right that the code generated for the critical mix_DT_4 seems to be identical. These .csv traces include a summary of the instructions executing so as to help you match them to your trees code. Then comparing the reported cache events per cycle, I can see:

- The verisim report added a "total cycles" column to the report.
- The TCS 5.01 version on verisim reports more I\$ misses as well as more D\$ misses.
- While the actual numbers reported in the columns are slightly different, the general pattern of which instruction causes many or few prefetches is similar.

From these comparisons, we can conclude that the same general results can be achieved with either version of the tools. The Verisim version of the verilated simulator is supported. The "verilator" version (eg sim5250) is not supported. We recommend that you use the newer version.

3.3 RUNNING ON THE HARDWARE

The example application exHwPrefetch runs this same algorithm on the hardware. It demonstrates how to observe the results using TimeDoctor and using your own monitoring of the cache counters. This version also introduces a component that becomes necessary in a real multitasking environment. If you are optimizing an H264 decoder, a certain set of prefetch settings can be found to be ideal. Imagine now that you are running two copies of the decoder. Since the prefetch regions are based on buffer addresses, the prefetch settings are likely to need to be different for each of them. The same would be true if you wanted to optimize an audio encoder. The settings need to be changed when you switch tasks. The component tmdlPrefetchRegion handles this. Its use is demonstrated in this example.

The example takes a "step" specification on the command line. Steps 1, 2 and 3 match those in the simulator demos. Step 4 adds prefetching of the output buffer, though you can see this does not help much.

The data is collected in two ways. One directly accesses the timers. This is nice because you control exactly what you are monitoring and when. The Time Doctor counters, on the other hand, only measure to task boundaries. This is good to know when you need to use Time Doctor to measure the performance of an algorithm in the context of a real application.

3.4 SUMMARY OF COMPARISON

How do the various measurement methods compare? Here are tables for the L2 misses and data cache stalls taken by the various methods

Table 8: Comparison of L2 D\$ miss counts

L2 Misses Verilator VerisimHW DirectTimeDoctor

Step 1	1306	1300	1579	1585
Step 2	678	675	995	1074
Step 3	860	844	668	706

Table 9: Comparison of \$ stall counts

Cache Stalls	Verilator	VerisimHW	DirectTime	Doctor
Step 1	427222	427622	384716	397400
Step 2	365974	365930	347704	371726
Step 3	287320	284571	258782	277092

We see here that the numbers follow very similar trends. They are not exact, but it is clear that the major improvements discovered by thinking about the algorithm mapping onto the cache architecture do work the same way on the hardware.

Note also that while you can look at well more than a dozen events counted at every cycle using the simulator, you can only look at a few events at a time on the hardware.

There are in fact 8 hardware counters. There may be other users of these counters. That is why only four are demonstrated in use.

Figure 5: TimeDoctor trace showing counters in example program

Figure 6: TimeDoctor trace with multiple tasks

4.0 STALLS VERSUS CONFLICTS

Note the "simple test" in the top level example. It dramatically illustrates of how massaging code just a little can lower the number of cache conflicts/stalls produced by an application. It shows a factor 5 reduction in cache stalls via some very simple code transformations. There are 3 small code fragments along with the measured number of cache stalls. Each modification of the code results in more than a factor 2 reduction in number of stalls. It is important to note that the entire gain is the result of a reduction in cache conflicts, not cache misses. There is no way to count these separately on the tm5250 core.

```
#define MEM_SIZE (16 * 1024)
```

This produces lots of conflicts (stalls) because of back to back load/stores as well as reading and writing the same address. Low number of L2 misses are observed due to the next sequential prefetch. But because the CPU is accessing sequentially and quickly, prefetched data is still causing stalls, but is not causing an L2 miss

```

for(i=0; i < MEM_SIZE; i++)
{
#define INDEX (i)
    mem[INDEX] = mem[INDEX] ^ (i & 0xff);
}
Cache counters before: L1D$ Miss 001, L2D$ Miss 00, D$Stalls 00403
Cache counters after: L1D$ Miss 281, L2D$ Miss 03, D$Stalls 32027

```

Next, reduce the number of data cache conflicts by writing back to a different address than you are reading from. L2 misses do not increase because of the "allocate on write miss" policy. L1 misses doubled, but total stalls are cut in half. There are still a lot of stalls because of back to back loads and stores.

```

for(i=0; i < MEM_SIZE; i++)
{
#define INDEX (i)
    mem2[INDEX] = mem[INDEX] ^ (i & 0xff);
}
Cache counters before: L1D$ Miss 001, L2D$ Miss 00, D$Stalls 00401
Cache counters after: L1D$ Miss 538, L2D$ Miss 03, D$Stalls 15551

```

You can further reduce the number of conflict stalls by introducing some "space" between loads and stores. Ideally, this "space" would be filled with useful instructions.

```

for(i=0; i < MEM_SIZE; i++)
{
#define INDEX (i)
    l = mem[INDEX] ^ (i & 0xff);
#pragma TCS_break_dtree
    mem2[INDEX] = l;
}
Cache counters before: L1D$ Miss 001, L2D$ Miss 00, D$Stalls 00401
Cache counters after: L1D$ Miss 538, L2D$ Miss 03, D$Stalls 06105

```

5.0 REFERENCES

The tm5250 databook is a key reference.

The TriMedia Simulator documents that come with TCS.

The training Power Point slides that are provided with NDK release 5.5.

The Power Point slides on HW prefetching that are delivered with this package.

6.0 ACKNOWLEDGEMENTS

Thanks to Maurice Penners, who provided the pioneering example.

Thanks to Jonathan Coxhead who wrote the task switching tools.

Thanks to Gene Pinkston, who discovered the information about conflicts.