

Week 5 Progress Report

Tim Russell

*Department of Electrical and Computer Engineering
University of Maine
Orono, USA*

tgrussell@eiu.edu

I. INTRODUCTION

The gist of my efforts was consolidation and evaluation of my progress thus far. As of last week, I had developed the general aspect of my greedy algorithm, determining its gross function and behavior. The largest unanswered question from last week was exactly what algorithm I would use to determine the victim block once the victim chip had been chosen. I still feel as if there are different gross directions I can take with the algorithm, as well as smaller refinements which could optimize the algorithm's behavior. Additionally, I have not conducted an in-depth analysis of the algorithm's complexity, preferring to wait until a more-complete product was ready. With some brief, back-of-the-envelope calculations, however, it is apparent that my algorithm is probably worst-case linear with respect to n , the number of requests.

Disconcertingly, I've run into a sort of algorithm writer's block. I feel as if I've had some good insights into the problem solution over the last couple of weeks and have tried to develop these ideas quickly. This last week found me in a rut, however, with respect to ideas about the solution algorithm. I met with Jianhui a few times over the week, and he offered constructive criticism and suggestions, but with little fruit on my part. I hope that by collecting my notes in the following report, I will at last find a little more direction.

My work this week mostly consisted of trying some run-throughs of request sequences to determine the algorithm's actual behavior. Along this line, I also created some tools to assist me in the evaluation process, which I will show below. I also attempted to create some metrics by which I could determine the algorithm's energy efficiency and performance trade-offs, and I will discuss these. Along the way, I did develop some insights into the behavior of the algorithm, which I can count as some benefit, and which I will present below.

II. ALGORITHM EVALUATIONS

To examine the behavior of my algorithm, I spent a while running through some cache replacement scenarios. I have become increasingly tired of drawing these access timelines over and over again (especially since I can't draw very well), so I decided to create some style sheets (figure 1). I also wrote a *very* simple program to generate random request sequences. Armed with these tools, I went through some request sequences, hoping to find holes in my approach and possibly generate new ideas for my algorithm. Specifically, I

studied the behavior of the algorithm when I allowed cache hits to be considered in the cache replacement decisions, versus looking at only cache misses. I compared these results to a simple alternative algorithm, LRU, to establish a baseline performance.

A. Evaluation Tools

To generate the aforementioned diagrams, I hoped to use a drawing program called *IPE*, a great tool which generates eps, pdf, and many other types of images. These images can have code embedded in them and can, conversely, be embedded in \LaTeX documents with ease. I also hope that *IPE* will be useful in my later work, especially when creating final reports, etc. My finished style sheet (figure 1) gives several timelines on which to sketch out memory accesses, and gives grid-boxes which can serve as placeholders for bookkeeping cache contents. Additionally, I also wrote an exceedingly simple C++ program, `randgen` (see figure 2) that would generate a pseudo-random cache configuration and then request sequence, given a few arguments.

B. Problems with Model

Neither of these two tools was very sophisticated, but they easily automated some of the more mundane details of my algorithm walkthroughs. A downside is that they made concrete some assumptions that I have been using in my earlier work. Specifically, the sheet suggests uniform time between requests and uniform size of requests. `randgen`, in turn, generates totally random block accesses. This is, of course, a simplification of an actual request pattern (and it made the program very easy to write), but conclusions that I may draw from my work may not be very descriptive of actual algorithm behavior, since looping access patterns and others could elicit very different behavior.

C. Conclusions

My main hope for these evaluations was to try to develop new ideas for my algorithm. Unfortunately, this was not the case, as I did not note many interesting points about the behavior of the algorithm. As mentioned, I compared the hit and no-hit methods. That is, I looked at the behavior of the algorithm when I did not project the access times for cache hits on the access timelines. In this way, I noted when a cache hit occurred, but when determining the victim chip in the case of a cache miss, the calculation of Δ_j never looked

at the completion times of cache hits, only cache misses. I compared this with the algorithm behavior when I *did* include cache hits into the determination, and I found no notable difference in hit ratios. The biggest difference between the two approaches is that when I did not allow cache hits to be used in the replacement decision, one memory chip tended to remain as the victim, very rarely or never escaping the role. Alternatively, when I did allow cache hits to be used in the replacement decision, different chips were selected as the victim chip at different times, though the shift from one victim chip to another was often slow.

Additionally, I did not see a significant performance penalty or gain as compared to LRU. In some initial cache configurations and access sequences, LRU performed better than my algorithm, while it seemed that LRU performed more poorly than mine for other randomly generated sequences. I'm sure that the relatively small number of accesses and random access patterns concealed true algorithm behavior of both, though.

III. DEVELOPING METRICS

I have met with Jianhui several times a week over the last few weeks, and in speaking with him last week, he noted the following. My algorithm, as formulated, seeks only energy-efficiency; it makes no concession for performance. This is true, and of course we cannot make a total sacrifice of performance for energy conservation. He mentioned that a possible approach to circumvent this stumbling block was to develop metrics that I could use to evaluate the energy efficiency and performance of the algorithm. I thought at the time he wanted me to be able to show how much more energy-efficient my algorithm was and how its performance fared, as compared to some other popular algorithm(s). As a result, the metrics I developed were necessarily generic and unimaginative. I wasn't sure how helpful they would be at this stage of my work, and I didn't spend long trying to delve into it.

I have since found out what (I think) Jianhui was trying to suggest to me, and it appears that I was right about my first approach. When we talked this week, after he heard some of the work I'd been going through, he asked why I was trying to evaluate my algorithm's performance. It seems that he was actually suggesting last week that I try to develop some performance tradeoff criteria. I didn't realize, for instance, that in his MRU/LRU approach, he was using LRU with a "window", by which he was limiting how long he would look for the LRU block in the MRU chip. I have been using an unmodified LRU approach in selecting victim blocks in the chosen chip, and I didn't realize some of the added complexities of the LRU approach (for instance, that a *global* LRU stack is maintained, meaning that it may not be quick to search for the LRU block of an arbitrary chip). His suggestion, was apparently, then, to develop criteria by which I could determine how much performance I was willing to trade for energy-efficiency, akin to his LRU "window."

So, my efforts in developing metrics have so far been unhelpful, but that is because I misunderstood the suggested

approach. I will try to develop this line now that I have a better understanding.

IV. REVELATIONS ON MD ALGORITHM

One benefit of the consideration that I have applied to my algorithm this week is that I have developed the following conclusions regarding it. Specifically, it has become apparent that the approach to choosing the victim chip is very similar to MRU, but I have noticed the following:

- 1) MRU looks at the *arrival times* of memory requests in making its determination. Contrastingly, MD looks at the *completion times* of memory requests. It has not escaped my attention that, if we consider only cache misses in making our determination of victim chips, and if θ_{dm} is constant, then MD is MRU. Indeed, we make the assumption that θ_{dm} is constant in our model, because it simplifies the problem, and because I have no idea how disk and memory accesses vary (or more importantly, how to reasonably incorporate such variation in my model). Still, in the "real world", this factor can and will vary, especially in the storage server environment, where multiple disks composed of heterogeneous hardware may be accessed and which may be in various power modes. So, this algorithm is at least nominally responsive to current hardware behavior.
- 2) If we do incorporate cache hits into our replacement decisions, it is easy to construct an example that shows that this algorithm does not choose the MRU chip.
- 3) Finally, MD chooses tends to choose as victims those chips which have longer DMA operation completion times. Hence, it favors chips which are missing, that is, those chips which have already recently experienced evictions. One consequence is that chips which are hitting are more easily able to keep their contents. I'm not yet sure whether this actually positively affects system performance, but it is a hope. I do, however, note that hitting chips may easily still have stagnant blocks which should, for performance enhancement, be evicted. It is easy to see that a chip which hasn't been scheduled as a victim recently might have old blocks which, because they are taking up space, are causing the missing chips to be victims more often.
- 4) This last point highlights a suggestion made by Jianhui recently, that I work on block placement so that I can enhance system performance by grouping the right series of blocks together in the same chip. This might allow concurrent access to multiple blocks while achieving energy savings, since requests in series will hit the same memory chip. I will try to look into this idea next week.

a_{16} a_{17} a_{18} a_{19} a_{20} a_{21} a_{22} a_{23} a_{24} a_{25} a_{26} a_{27} a_{28} a_{29} a_{30} a_{31} a_{32} a_{33}

a_{16} a_{17} a_{18} a_{19} a_{20} a_{21} a_{22} a_{23} a_{24} a_{25} a_{26} a_{27} a_{28} a_{29} a_{30} a_{31} a_{32} a_{33}

a_{16} a_{17} a_{18} a_{19} a_{20} a_{21} a_{22} a_{23} a_{24} a_{25} a_{26} a_{27} a_{28} a_{29} a_{30} a_{31} a_{32} a_{33}

a_{16} a_{17} a_{18} a_{19} a_{20} a_{21} a_{22} a_{23} a_{24} a_{25} a_{26} a_{27} a_{28} a_{29} a_{30} a_{31} a_{32} a_{33}

a_{16} a_{17} a_{18} a_{19} a_{20} a_{21} a_{22} a_{23} a_{24} a_{25} a_{26} a_{27} a_{28} a_{29} a_{30} a_{31} a_{32} a_{33}

a_{16} a_{17} a_{18} a_{19} a_{20} a_{21} a_{22} a_{23} a_{24} a_{25} a_{26} a_{27} a_{28} a_{29} a_{30} a_{31} a_{32} a_{33}

$k = 4$
 $b = 4$
 $n = 33$
 $j = 4$
 $\tau =$

Fig. 1. Form sheet for algorithm walkthroughs

```

#include<stdlib.h>
#include<iostream>
#include<time.h>

using namespace std;

int main(int argc, char **argv)
{
    srand(time(NULL));
    if (argc == 4)
    {
        int numreps = atoi(argv[1]);
        bool found = false;
        int hitCount = 0;
        int tempVal = 0;
        int range = atoi(argv[2]);
        int numBlocks = atoi(argv[3]);
        int cache[numBlocks];
        int i = 0;
        while (i < numBlocks)
        {
            tempVal = (rand() % range) + 1;
            for (int j = 0; j < i; ++j)
            {
                if (tempVal == cache[j])
                {
                    found = true;
                    break;
                }
            }
            if (!found)
            {
                cache[i] = tempVal;
                ++i;
            }
            found = false;
        }
        found = false;
        cout << "Cache contents: ";
        for (i = 0; i < numBlocks - 1; ++i)
        {
            cout << cache[i] << ", ";
        }
        cout << cache[numBlocks - 1] << endl;
        for (; i < numreps; ++i)
        {
            tempVal = (rand() % range) + 1;
            cout << '[' << i + 1 << "]: " << tempVal;
            cout << endl;
        }
    }
    else
    {
        cout << "Parameters not specified." << endl
            << "usage: randgen <number of outputs> <range of output [1,2...,range]> "
            << "number of memory blocks" << endl;
    }
    return 0;
}

```

Fig. 2. Code for randgen program