

Week 7 Progress Report

Tim Russell

Department of Electrical and Computer Engineering
University of Maine
Orono, USA
tgrussell@eiu.edu

I. INTRODUCTION

This week marked a foray into yet another variation on one of the central ideas in my algorithm design—overlapping DMA access. We investigated a new approach to the problem, utilizing the idea of making our cache *replacement* policy act more as a *placement* policy. This approach is actually an amalgamation of the last two approaches. The *Magic Dragon* Algorithm aggressively pursued replacement options which maximized DMA overlapping, while the *Slippery Lizard* Algorithm looked to make intelligent placement decisions to promote future DMA overlapping. I will talk briefly about the *Omnivorous Ocelot* algorithm in section II.

A main thrust of this week was an attempt at evaluating the efficacy of this approach. In pursuit of that evaluation, I wrote a program, *analyzer*, which took real-world disk traces and implemented the algorithm in a simplified, abstracted manner (see section III-A). The goal was to see if we could achieve the kind of DMA overlapping for which we were looking. The program by itself only generated a great deal of intermediate numerical data which is, of course, not an easily digestible medium for humans. So, in order to actually evaluate the viability of the algorithm, I wrote a couple of MATLAB functions—which I will talk about in section III-B—to transition the raw data into visual form.

II. Omnivorous Ocelot ALGORITHM

A. Developing Correlations

We find ourselves again appealing to the idea of energy conservation via DMA concurrency. If we, in our omniscient, *offline* knowledge could intelligently place blocks which are consecutively (or nearly so) accessed in the same memory chip, we could achieve DMA overlapping. This was a goal of *Slippery Lizard*, but the actual development of such an approach proved difficult. Indeed, even in the *offline* case, trying to decide how to make time- and space-complexity tradeoffs became difficult and convoluted. The *Slippery Lizard* eventually birthed a large Filter Matrix which helped generate an intermediate Correlation Matrix which then helped identify a Main Matrix which generated a Group Array and an associated Group Membership binary search tree, which then ..., you get the point.

The step forward this week was in making a useful and powerful simplification; we assume that all requests in the request sequence are *cache hits*. We can generate group

membership (or *temporal correlation*) as mentioned before. For a given pair of requests A and B , suppose that the arrival time of A is before that of B . We can compute the amount of time that the two requests overlap with $CT(A) - AT(B)$, where $CT(X)$ is the completion time of the request for block X , and $AT(X)$ is the arrival time of the request for block X (shown below in figure 1).

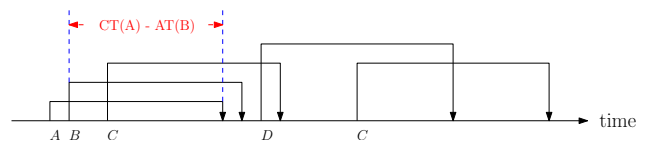


Fig. 1. Calculating Block Requests' Overlap

We perform the computation for all blocks in the request sequence and remember the total amount of time that any two blocks overlap—in figure 1 this would entail computations for (A, B) , (A, C) , (B, C) , and (C, D) (twice, and adding the results). By doing so, we generate a weighted, undirected graph $G(V, E)$, in which V is the set of all unique block numbers which are requested in a given request sequence. Further, there is an edge $e(AB) \in E$ if A and B overlap in the described way, and the weight $w(AB)$ of $e(AB)$ is the total amount of overlap time between A and B (a weight of 0 is implied if there is no edge). From figure 1, we would generate a graph like the following:

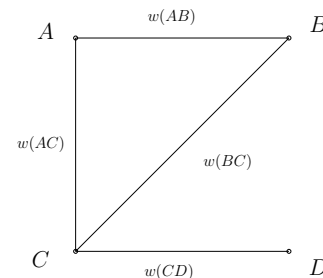


Fig. 2. Correlation Graph

B. Decision Making

Once we have a generated a graph similar to figure 2 (though much larger and more complex, of course) for the entire request sequence, we can use it in the following manner. Suppose that all memory chips are full, and that at time t_i

block b_i is requested, and it is a *cache miss*. Then we can go through each memory chip and sum the weights of the edges between b_i and each block currently in the memory chip. The chip with the highest sum contains blocks which, in toto, have the greatest temporal correlation with b_i . We would have the greatest chance at energy savings via DMA overlap during future requests if we placed b_i in that memory chip; hence, that memory chip is chosen to be the victim chip. That is, for a cache C , composed of m memory chips—so that the memory chips are labeled C_j , $j \in \{0, 1, \dots, m-1\}$ —then the victim chip, v , is chosen by:

$$v = \max_{C_j | j \in \{0, 1, \dots, m-1\}} \left(\sum_{x | x \text{ is a block} \in C_j} w(b_i x) \right) \quad (1)$$

III. ANALYZING THE *Ocelot*

A. Generating Data From Data

To attempt to determine the feasibility of this approach, we desired to run a sort of pre-simulation which would show whether requested blocks actually exhibited the kind of temporal association of which we want to take advantage. Indeed, this approach relies entirely on the crutch of repeated temporal locality, and if it is not the case that such behavior occurs frequently enough and to a varied enough degree in actual traces, we should forego any more work along this line.

So, I wrote a C++ program `analyzer` which performed the most significant number crunching for us. `analyzer` takes one mandatory parameter (a trace *filename*) and one optional parameter (the maximum size of an array, discussed below). The program first reads in the trace file line-by-line, which consists of two integral values per line, a block number and the request time in nanoseconds. Each block number is added to an `STL map`, with the `key` being the block number and the `value` being the frequency.

We take the time to build this list of frequencies because of space limitations. The most simplistic (hence easiest to code) data structure to represent a graph like figure 2 is a two-dimensional square array. For a request sequence of n blocks, of which u are unique block numbers, I would need to allocate space for u^2 values, with the worst case being that every block was unique; hence my space complexity was $\mathcal{O}(n^2)$. I ran my program on three trace files: `db2_parallel_lbn_time.txt`, `tpch_time_lbn.txt`, and `tpcr_time_lbn.txt`. These files had 3756956, 14275597, and 10986993 requests, respectively. Looking at the smallest, of the 3756956 requests in `db2_parallel_lbn_time.txt`, approximately 240000 were unique. To build the correlation graph for all blocks require allocating an array which could store $5.76 * 10^{10}$ integral values. Supposing that we use 4-byte integers (smaller integer values like `short` don't have a large enough range), this require more than 214 GB of memory. If we first build the `map` of frequencies, we can pare this number down to our liking in the following manner. Starting at frequency $f = 1$ (i.e. blocks which were requested

only once), `analyzer` iterates through the `map`, deletes all blocks which have frequency f , increments f , and continues until the size of the `map` is not greater than the user-specified maximum array size (default is 1000). In this way, we control the size of the array, and we are able to filter out blocks with low frequencies. This second point is an added benefit, because we should not want to base our replacement decisions on blocks which occur only a few times, anyway.

Once we have reached the desired number of high-frequency block numbers, `analyzer` dynamically allocates arrays to hold the correlation information for all the blocks. With the preparation complete, the temporal overlap is calculated for all blocks in the trace file (the file has to be re-read at this point). As each request is read in, the block is checked to see if it is a high-frequency block (i.e. it is still in the `map`). If it is not, it is skipped; if it is in the `map` its overlap values are calculated as in figure 1. We use a constant cache hit completion time of 2000 nanoseconds in the calculations, a value that was derived as a usable average from trace analysis. Overlap values are computed for all blocks in the trace which are stored in the `map`, and the corresponding values are tabulated in the array. After finishing the overlap calculation, `analyzer` prints this information to a file and then forms some model groups to illustrate the maximum-correlation groups possible; these last are printed to screen.

B. Making Pictures

Since nobody wants to look at 5000×5000 arrays of unsigned `int`'s and try to derive usable information from that (or at least I don't), I moved on to the next tool in the kit, MATLAB. First, I (with the assistance of Andrew) wrote a MATLAB script, `lolpic`, which read in the table of temporal overlap values generated by `analyzer` and created a grey-scale image from the table. Specifically, each element of the table could correspond to a pixel on the screen, with the pixel being black if the corresponding array value was equal to the lowest value in the table, white if it was equal to the highest value, and some intermediate grey if in between. A problem with this visualization was that the largest value in the table was always dramatically larger than most other values. To wit, the lowest value was always 0, the average value might be approximately 200, and the highest value might be 657,000,000 (again, these values are the total number of nanoseconds that a pair of blocks overlapped). The effect of this vast range of values was that, by comparison, the large majority of table values were dwarfed by the magnitude of the largest value, meaning that most of the grey-scale values were very near the black end of the scale. Normalizing the values didn't help, because the relative disparity still existed. I used the MATLAB function `imshow` to try to adjust the grey-scale range to tighter ranges of values, and this helped a little.

I then wrote another MATLAB script, `commgraph.m`, which generates cummulation graphs from the frequency information generated by `analyzer`. Cummulations graphs show the request distribution of the trace, which often follows the Zipf distribution. We see in figure 3 that the

db2_parallel_lbn_time.txt trace did not, in fact, exhibit this distribution; it actually appears that this request sequence was almost uniform. The other two traces, shown in figure 5 and figure 7, show that the tpch_time_lbn.txt and db2_parallel_lbn_time.txt, respectively, do more closely resemble the Zipf distribution. From these two graphs, we see that both distributions have sharp changes in slope at clearly defined points. This shows that these traces did not have requests whose frequencies varied over a wide range; instead, subsets of blocks tended to have identical request frequencies, with sharp changes in the frequencies of these subsets. Additionally, figures 6 and 8 seem to show that the temporal correlation between blocks was locale-specific. The data are organized in the table in ascending order by block number, so that closely-numbered blocks are closer together in the graph. We see then, that there is a sharp change in these two figures between blocks requested in a certain physical area to another, disjoint, area.

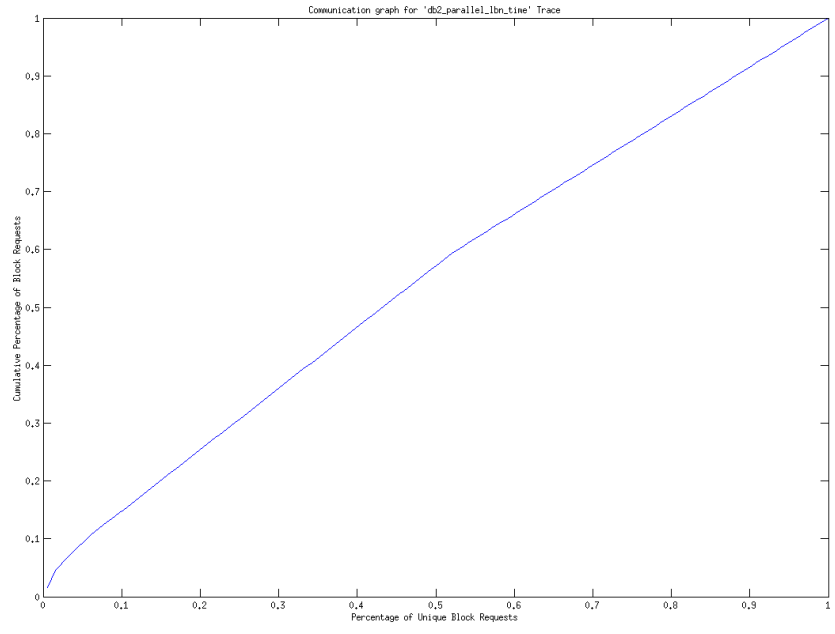


Fig. 3. Cummulation Graph 1



Fig. 4. Temporal Correlation Map 1

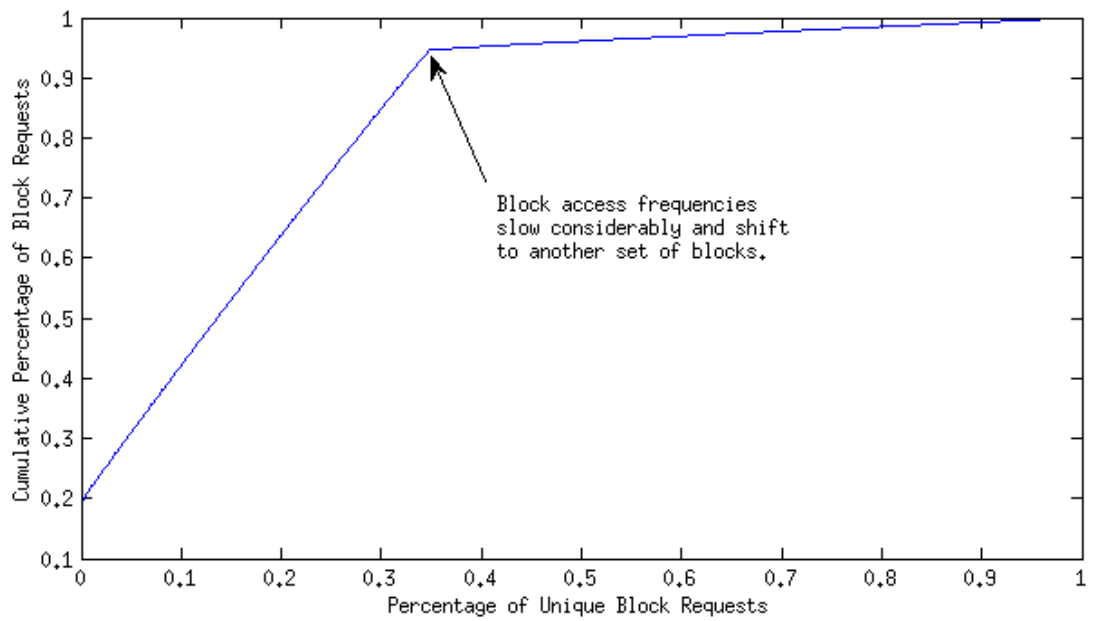


Fig. 5. Cummulation Graph 2

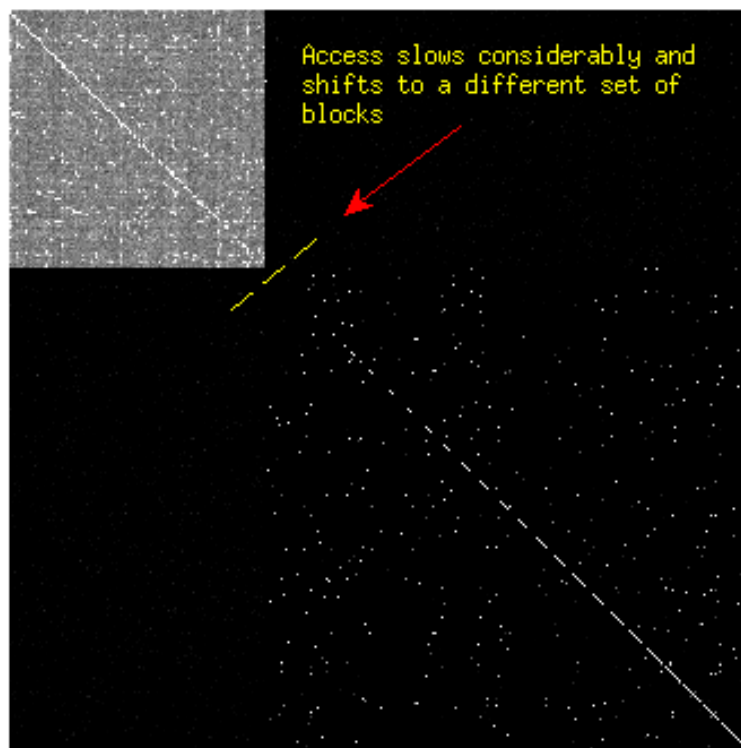


Fig. 6. Temporal Correlation Map 2

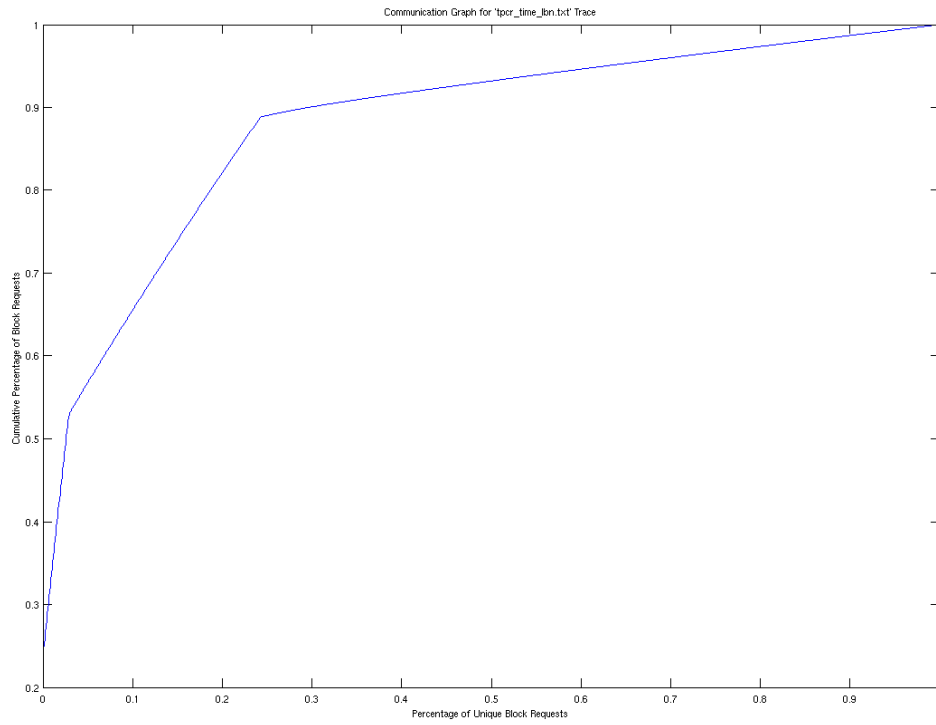


Fig. 7. Cummulation Graph 3

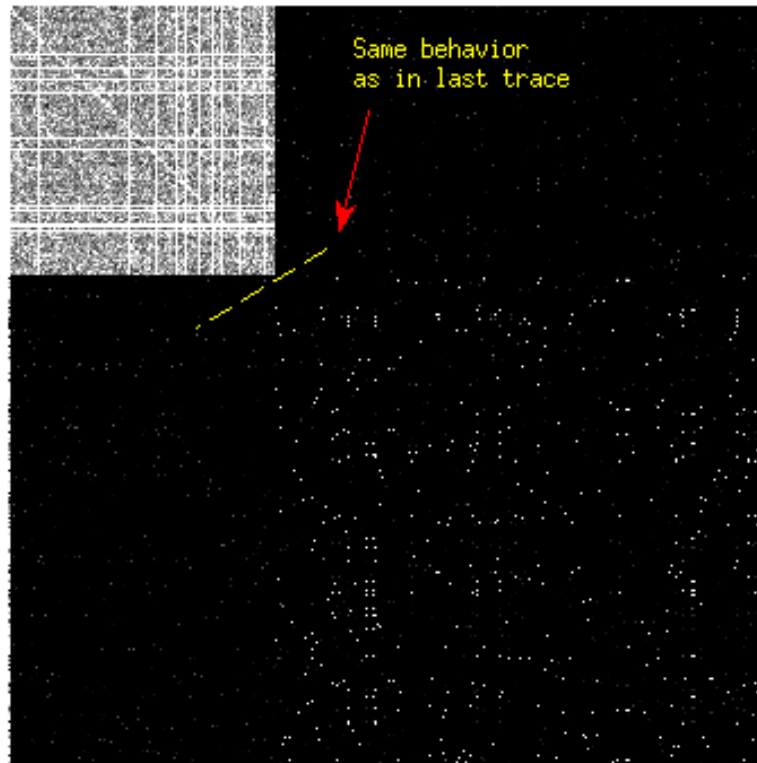


Fig. 8. Temporal Correlation Map 3