Appendices of Optical Overlay NUCA: A High Speed Substrate for Shared L2 Caches

Eldhose Peter, Anuj Arora, Akriti Bagaria and Smruti R. Sarangi

APPENDIX A
ALGORITHM TO BUILD THE OVERLAY

The overlay can be created either once in an application or periodically. First let us consider a system where the process of creation of the set of overlays happens only once for an application. In this case instead of dedicating special logic in hardware to build the overlay, for each bank we write its access count (number of times it has been accessed) to a dedicated location in memory. The hardware invokes a custom software routine, or a firmware routine to compute the overlays as shown in Figure 5. Each bank contains a list of all the other banks in its overlay, and also a list of its neighbors. The details of the overlay are saved in the structure *Overlay Info Store* (see Section 4.2).

Algorithm 1: OVERLAY-BUILDER returns $3n/2k$ logical sets of banks

Input: A set $A = \{a_1, a_2, \ldots, a_n\}$ of banks sorted based on the number of accesses ; $k \rightarrow$ the number of banks in each set, $n \rightarrow$ total number of banks
Output: $3n/2k$ logical sets of cache banks

```
/* Create hybrid overlays                                   */
for i ← 1 to n/k do
    BankSet_i[L, H] ← selectKBanks(A) ;
    OSV.add(makeStructure(L,H));
i ← a_1;
/* Create infreq overlays                                   */
while i ≠ n/k do
    L1 ← BankSet_i.get(L);
    L2 ← BankSet_{i+1}.get(L);
    OSV.add(makeStructure(L1,L2));
    i ← i + 2;
```

We show a generic algorithm (Algorithm 1), which creates $3n/2k$ overlays of cache banks, where k is the number of banks in each bank set and n is the total number of banks. In our system, $n = 32$, and $k = 8$ ($3n/2k = 6$). We thus have 6 overlays.

Let us now explain the operation of the algorithm shown in Figure 1. $BankSet_i$ denotes the i^{th} bank set. The function, *selectKBanks*, returns $k/2$ high-access banks (set H), and $k/2$ low-access banks (set L). The *makeStructure(A,B)* function creates an overlay by combing cache banks from sets A and B and adds them to the OSV(Overlay Structure Vector). First we create n/k *hybrid* overlays. Then we take the low accessed $k/2$ banks each from two adjacent *hybrid* overlays, combine them and create an *infreq* overlay. There will be a total of n/k *hybrid* overlays and $n/2k$ *infreq* overlays. In our case, there will be 4 *hybrid* and 2 *infreq* overlays.

• *All the authors are affiliated with the department of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi - 110016. E-mail: {eldhose, mcs132541,srsarangi}@cse.iitd.ac.in, anuj@cisco.com*

APPENDIX B
HEURISTICS FOR HOME BANK IDENTIFICATION AND OVERLAY CREATION

In this section we address two points: 1) determining the home bank from a memory address, and 2) need to create a complex overlay.

B.1 Selection of Bits to Identify the Home Bank

Let us assume a multicore processor with n banks. We have $n = 32$ banks in our processor, thus we require 5 bits to uniquely map an address to a home bank. The task is to select 5 bits in a 64 bit address for identifying the home bank. Let us first take out the bits to identify a byte in a cache line, which are always the least significant bits. If we consider 64 byte blocks, then we have a 58 bit block address. Now, we have several options as shown in Figure 2.

Let us first discuss the scheme, *end*. Here, we use the least significant n bits of the block address to identify the home bank. We expect these bits to have the maximum amount of randomness and thus this scheme is the most likely to yield the most uniform bank access distribution. Let us now consider another scheme called *Mid*. Here, we first remove the bits that are needed to identify a set (referred to as the *set index* bits). In our case, the number of set index bits are 8. We assume that the set index bits are in the least significant positions of the block address. We remove them, and then use the n LSB bits of the remaining part of the address to determine the bank. This scheme is expected to have a lesser degree of randomness than the *End* scheme.

We conducted simulations to record the access frequencies of different home banks. Note that in these simulations we use the S-NUCA configuration (no migrations/evictions across banks). In Figure 1 we show

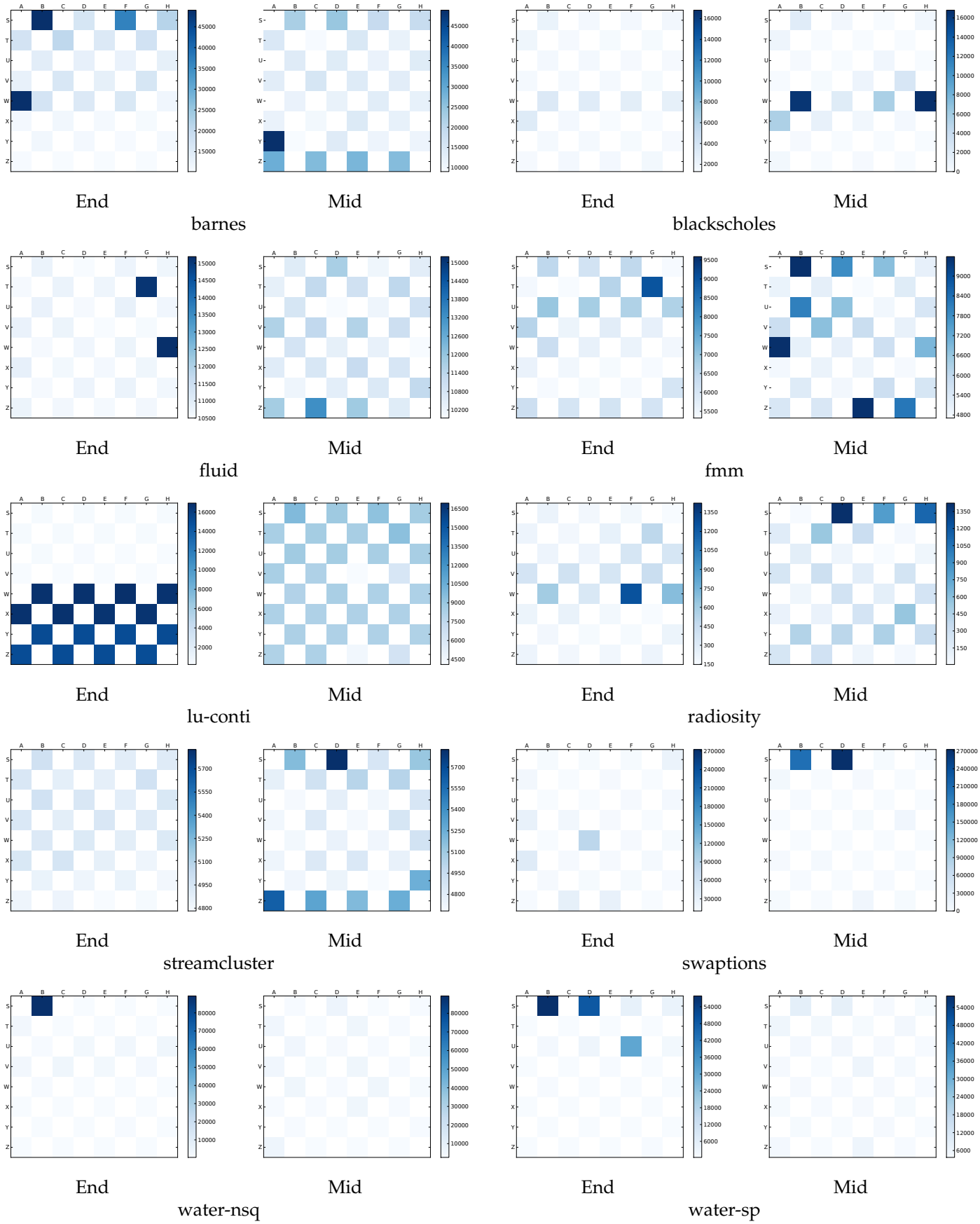

Fig. 1: Access pattern - Left side:- End Scheme, Right side:- Mid Scheme

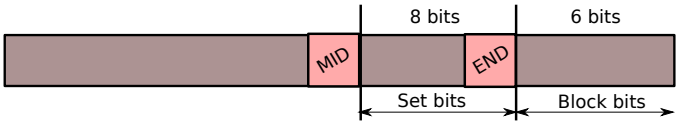

Fig. 2: Bit Selection

the distribution of access frequencies in a pictorial form. The deeper the color, the higher is the access frequency. We can see both cores and cache banks in the figure in a chess board like format. Note that the number of accesses for every core is zero.

From Figure 1 we cannot conclude that the *End* scheme generates a more uniform access pattern. For 3 out of 10 benchmarks, the access frequencies have roughly the same pattern. In the case of *barnes*, *lu-cont*, *water-nsq* and *water-sp* the *Mid* scheme shows more uniformity while in the case of *blakscholes*, *streamcluster* and *swaptions* the *End* scheme shows more uniformity.

However, the even stronger fact that we can conclude is that access frequencies are genuinely non-uniform even for the *End* scheme. The access frequencies routinely vary by more than 4X across banks. In some cases they can even vary by up to 1000X. This observation justifies our choice of creating overlays based on variance in access frequencies.

B.2 Simple vs Complex Overlays

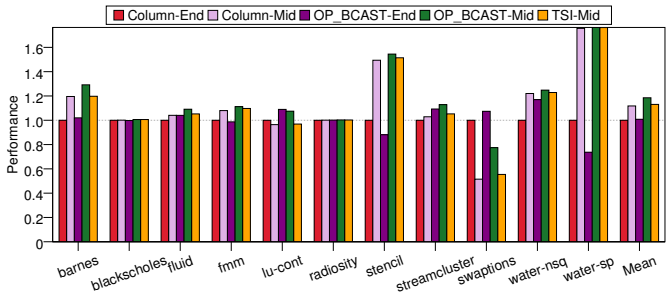

Fig. 3: Performance

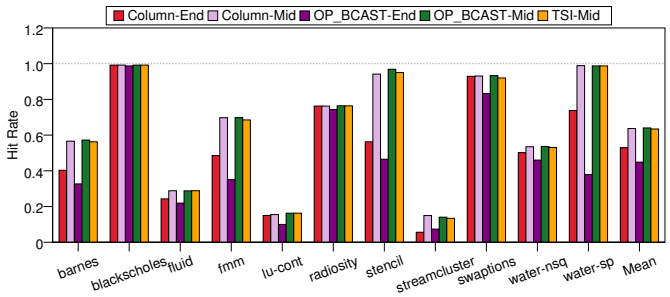

Fig. 4: Hit Rate

The effectiveness of the optical overlay is an important assumption that needs to be proved. We have to analyze the need for creating a complex overlay instead

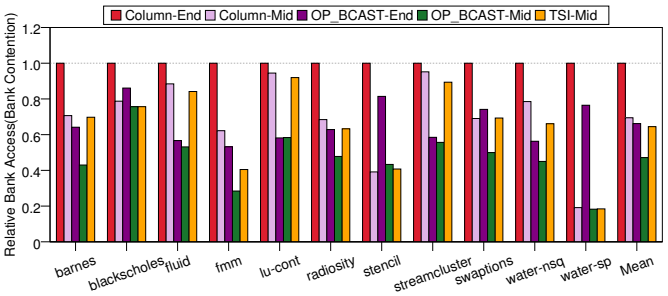

Fig. 5: Accesses

of using a simple and static set of cache banks such as D-NUCA. We could have taken a proximate set of cache banks as a bank set and used the same optical network for communication. Recall that D-NUCA uses such an approach where it treats all the banks in a column as a bank set. Let us call this configuration, *Column*, where we statically assign the cache banks in a column to a bank set. In comparison, in our proposals, we use a bank set based on the bank access frequencies of the benchmarks(*OP_BCAST*).

In Figure 3, we compare the performance of different combinations – Column-End, Column-Mid, *OP_BCAST*-End, *OP_BCAST*-Mid and TSI-Mid, normalized to Column-End. We can see that *OP_BCAST*-Mid performs better than all the other configurations. It shows an improvement of 30-70% in *barnes*, *water-nsq* and *water-sp*. In all our configurations, the *Mid* configurations are better than the *End* configurations. For example, Column-Mid is better than Column-End by 12% (on an average).

In Figure 4, we compare the L2 cache hit rates. We can see that Column-Mid and *OP_BCAST*-Mid have the best hit rates. On an average, the hit rate of *OP_BCAST*-Mid is 20% and 10% better compared to Column-End and *OP_BCAST*-End respectively. Note that Column-Mid and *OP_BCAST*-Mid have roughly similar hit rates. The difference in performance comes in latency, which we shall show next in Figure 5.

Figure 5 shows the number of cache banks accessed, and is thus indicative of bank contention. The access numbers are normalized to Column-End. Here we can see that the number of accesses by *OP_BCAST*-Mid is almost 50% compared to Column-End. This leads to reduced traffic and contention, thus explaining the superior performance of *OP_BCAST*-Mid.

From these Figures, we can conclude that by creating a set of banks based on the program characteristics helps in improving the performance. Also the *Mid* scheme performs better compared to the *End* scheme.

APPENDIX C
EXTENDED RESULTS

In the main paper, we have shown the results for a subset of benchmarks due to limitations in space. In this section we show the results for all the benchmarks

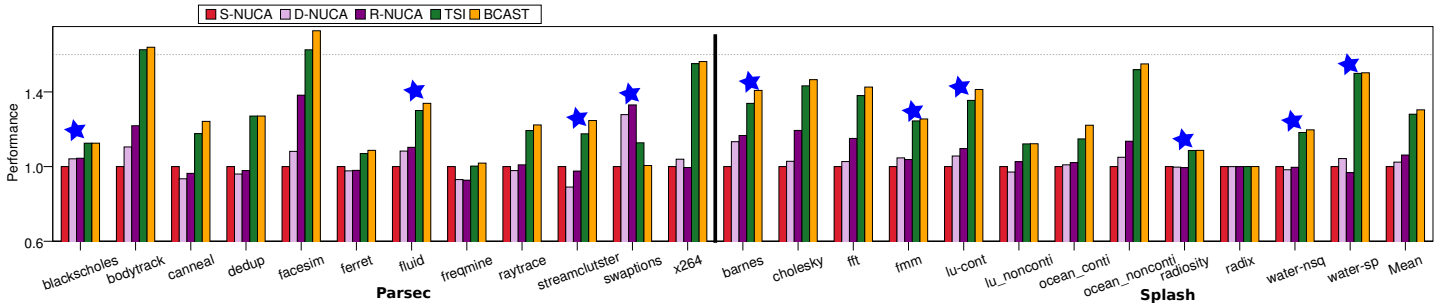

Fig. 6: Instruction throughput comparison of all benchmarks

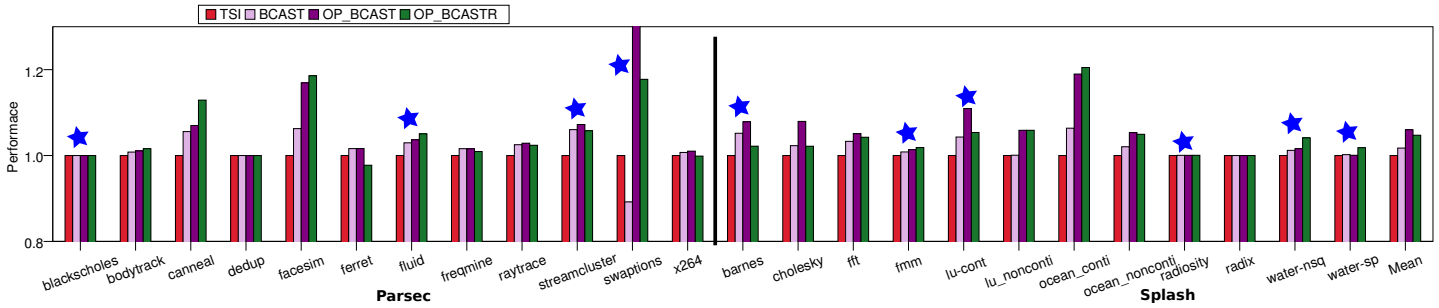

Fig. 7: Instruction throughput comparison of all benchmarks

in the Parsec [1] and Splash [2] benchmark suites. The results follow the same trend as the results shown in the evaluation section. The selected subset contains the benchmarks that perform almost the same as the entire set of benchmarks, when we compare the mean performance. In Figures 6 and 7, we show the performance results for all the benchmarks, including the benchmarks shown in the main evaluation section. The benchmarks selected in the main paper are indicated with a star.

REFERENCES

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *PACT*, 2008.

[2] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, 1995.