Optimizing Matrix Operations Using Novel DRAM Access Primitives

CMU 15-745: Optimizing Compilers (Spring 2015)

Final Report

Joy Arulraj (jarulraj), Anuj Kalia (akalia), Jun Woo Park (junwoop)

1 Abstract

Traditionally, when an application requests a set of different rows residing on the same memory bank, the memory access latency is increased due to DRAM row-buffer conflicts. These row-buffer conflicts can be reduced by interleaving consecutively addressed data locations across memory banks, so that the requests are spread across different banks. Software systems, therefore, strive hard to lay out their data structures to generate access patterns that can be served efficiently by the DRAM chips. For instance, DBMSs targeting analytical workloads use a columnar data layout so that values in the same tabular column are stored in consecutively addressed data locations [1].

The design complexity of software systems can be significantly reduced in future DRAM chips using novel access primitives that can support both row and columnar access patterns efficiently. This is feasible by appropriately laying out the data and transforming the DRAM access methods. In this project, we focussed on automatically transforming the software access patterns to leverage these DRAM access primitives during compilation time. We developed a DRAM access cost model to select the appropriate DRAM access primitive for different patterns. We then analyze the software access patterns in our compiler pass and then transform the code to make use of these access primitives. Our evaluation of the performance impact of these transformations in a matrix library showed that we can realize 2-5× speed-up out-of-the-box for the same tasks by choosing the optimal DRAM access pattern.

2 Introduction

DRAM access patterns significantly impact the performance of memory-bound applications. The delay incurred while handling a cache miss is not constant as the access latencies within DRAM are not uniform [11]. DRAM is organized as a set of banks, each of which is made up of multiple chips. Consecutive memory addresses are interleaved evenly across the DRAM banks. Each bank has a row-buffer that is used to speed up DRAM accesses. When the application requests for a page in a bank, the page is first fetched into the bank's row buffer. Then, it is sent to the the application. If the application happens to request for a page that is already present in the row buffer, it is termed as a *row-buffer hit*. This results in the minimal DRAM access latency i.e. one memory cycle. However, if the page is not present in the row buffer, then the appropriate row in the DRAM bank needs to be activated before the requested page can be loaded into the row buffer and returned to the application. This event is referred to as a *row buffer miss*.

Programmers strive hard to lay out their application data structures to generate access patterns that can be served efficiently by the DRAM chips. For instance, DBMSs targeting analytical workloads use a *pure-columnar* data layout so that values present in the same tabular column are stored in consecutively addressed data locations [1, 9, 2]. This helps spread the DRAM accesses to multiple DRAM banks that can serve the requests in parallel. Thus, the application can better utilize the available DRAM bandwidth.

To give a concrete example, we compare the average time taken to compute the sum of a row and sum of a column in a matrix laid out in row-major order. The results are shown in fig. 1. We observe that summing a row in this case is 2–4× faster than the columnar sum operation.

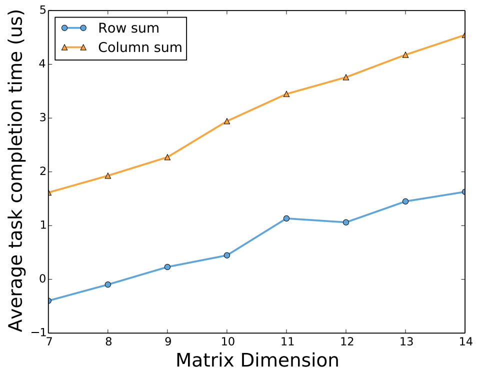


Figure 1: Performance comparison of row and column oriented access patterns. The dimension attribute along the x-axis refers to the $\log_2$ of the length of the two-dimensional square matrix.

2.1 The Problem

In this work, we focus on applications that have complex workloads with mixed DRAM access patterns. For instance, in a hybrid transaction and analytical processing (HTAP) workload [6], the DBMS needs to access both the tuples and columns of a table efficiently. Irrespective of whether the DBMS uses a pure-row or pure-columnar layout, one of the two access patterns has to suffer from poor DRAM performance. For instance, assume that the DBMS uses a pure-row layout and that the size of each tuple is one cacheline (64 bytes). When serving an analytical query that needs to sum up the values in a particular column, we need to access only that part of the tuple in all the tuples in the table. This means that we would like to access only part of a cacheline. This is however not feasible in current hardware, as the data is laid out in tuple-major order.

In future hardware systems, we anticipate that we will have new DRAM access primitives that allow us to access parts of a cacheline and not just entire cachelines. These primitives will allow the application to read a cacheline composed of data stored in different DRAM rows by specifying an DRAM *access pattern*. This is illustrated in fig. 2. Consider a 4×4 toy matrix. In the default row-major layout, fetching the first column will require four DRAM read operations as they will be serialized at the same bank. If we *shuffle* the tile to DRAM row mapping as shown below, we observe that we can access both the first row (cacheline 1) and first column (denoted by, say, cacheline 1.2) using a single DRAM read operation. This effectively allows us to retrieve the required data (for instance, a column in a matrix) in fewer DRAM operations compared to existing DRAM access primitives. Therefore, we can achieve (1) lower DRAM latency, (2) more efficient DRAM bandwidth utilization, and (3) lower energy consumption.

The problem we address in this work involves designing a compiler pass that allows us to *automatically* identify the application's access patterns and transforms the program to make use of these novel DRAM access primitives. This reduces the burden on the programmer and thereby simplifies the adoption of these DRAM access primitives.

2.2 Our Approach

At a high level, we decouple the problem into two subproblems:

- We need to figure out the different application access patterns
- We then need to map the application access patterns to make use of underlying DRAM access primitives.

We have implemented an LLVM pass that analyses data structure access patterns and uses a DRAM access cost model to map the access pattern to an optimal DRAM access primitive. Our pass can successfully handle arbitrary data structure access patterns over multi-dimensional arrays of primitive data types as well as structures.

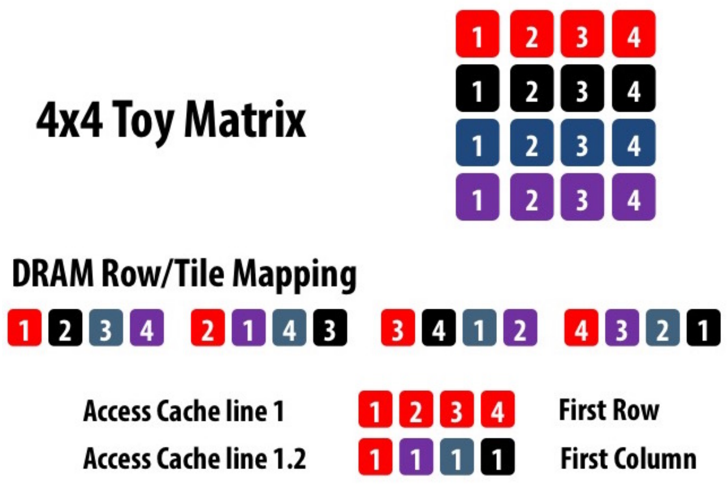


Figure 2: Novel DRAM access primitives that allow the application to read both a column or a row in a small matrix in a single DRAM read operation.

2.3 Related Work

We have gone through several traditional papers on DRAM access primitives and data locality [11, 7, 10]. As the DRAM primitives we are targeting in this work are not currently available, we could not find more relevant literature. However, there has been a lot of prior work on handling hybrid application patterns with existing hardware constraints, particularly in the area of database systems. Copeland et. al. [4] presented the advantages of a storage model based on decomposition over conventional row-oriented storage models. More recently, HYRISE [6] and MonetDB/X100 [3] systems have been designed to handle hybrid and analytical query workloads respectively on existing memory subsystems. In particular, the X100 engine [12] introduced in-cache vectorized processing to balance between the column-at-a-time execution in MonetDB and the traditional tuple-at-a-time Volcano pipelining model.

2.4 Contributions

We make the following contributions in this project:

- We implemented an LLVM pass that automatically determines the different data structure access patterns by an application.
- We developed a weighted DRAM access cost model that allows the compiler to map the application access pattern to the most optimal DRAM access primitive.
- We illustrate the performance benefits of these novel DRAM access primitive through a quantitative evaluation on a matrix library.

3 Memory Access Pattern Analysis

The goal of this analysis is to identify the different application access patterns and then map then onto an efficient DRAM access primitive to improve cache and memory subsystem performance. In this section, we will first present the compiler pass and then describe the DRAM access cost model that we leverage in our mapping function.

3.1 LLVM Pass

We define an application's data structure access pattern in terms of two parameters : (1) *stride*, and (2) *offset*. For instance, consider an array of structures, each of which contains two integer fields. Then, if the application accesses the second field in all the structures in the array in a loop, we denote that access pattern as $\{4, +, 8\}$. Here, the first member is the offset and the second element is the stride.

We implemented a compiler pass that analyses our matrix library automatically to determine the stride and offset of the application's access patterns. We assume that the programmer annotates the data structures that should be analysed using LLVM annotations. In theory, we can apply our analysis on all loads and stores in the program. However, we wanted to restrict our analysis to important data structures as that approach is more practical. Further, we also focus only on memory accesses within *loops* in the application. This is because we expect that optimizing these memory accesses with the novel DRAM access primitives will be sufficient to realize significant application speedups. This compiler pass consists of three stages :

- We first parse the programmer annotations to determine the critical data structures.
- Then, for each loop in the program, we analyze the loads and stores associated with these data structures to determine the access patterns.
- Finally, we use the cost model to map the access pattern to the relevant DRAM access primitive.

We implemented the first stage using LLVM's support for data member annotations. Then, we leveraged the scalar evolution (SCEV) analysis [8] functions in the second stage of the compiler pass. These functions support the analysis of scalar expressions within loops and help recognize general induction variables. Given a program *scope* and a register, we can use these functions to obtain an expression that represents the register's evolution inside the scope. We applied these functions to the load and store addresses of the annotated data structure within the scope of the innermost loop. We then parsed the recursive expression and delinearized it to obtain the stride and offset with which the application accesses the critical data structures in the loop.

The output of the compiler pass on a sample program is shown below :

Listing 1: Sample Input Program

```
------------------------------------------------------------
INPUT PROGRAM
------------------------------------------------------------

void foo(long n, long m)  {
  __attribute__((annotate(‘‘this is important’’))) int A[n][m];

  struct key{
    char a;
    char b;
    char c;
  };
  __attribute__((annotate(‘‘critical’’))) struct key keys[100];

  char x;
  for (long i = 0; i < n; i++) {
    for (long j = 0; j < m; j++){
      A[i][j] = 0;
      A[j][i] = 0;
      x = keys[i].a;
      keys[i].b = x;
    }
  }
}
```

Listing 2: Analysis Output

```
------------------------------------------------------------
ANALYSIS OUTPUT :
------------------------------------------------------------


Analysing function :: foo:

------------------------------
Annotations found :
------------------------------

tests/a.c:i32 5
%vla = alloca i32, i32 %1, align 4
this is important

tests/a.c:i32 13
%keys = alloca [100 x %struct.key], align 1
critical

------------------------------
Loads/Stores :
------------------------------

------------------------------
Instruction:   store i32 0, i32* %arrayidx6, align 4
```

```
------------------------------

Access Pattern :
{{0,+,(4 * %m)}<%for.cond>,+,4}<nw><%for.cond3>

Delinearized Access Pattern :
[{0,+,1}<nuw><nsw><%for.cond>][{0,+,1}<nuw><nsw><%for.cond3>]

------------------------------
Instruction:   store i32 0, i32* %arrayidx8, align 4
------------------------------

Access Pattern :
{{0,+,4}<nuw><nsw><%for.cond>,+,(4 * %m)}<%for.cond3>

Delinearized Access Pattern :
[{0,+,1}<nuw><nsw><%for.cond3>][{0,+,1}<nuw><nsw><%for.cond>]

------------------------------
Instruction:   %4 = load i8* %a, align 1
------------------------------

Access Pattern :
{0,+,3}<nuw><nsw><%for.cond>

------------------------------
Instruction:   store i8 %4, i8* %b, align 1
------------------------------

Access Pattern :
{1,+,3}<nw><%for.cond>
```

In the listing above, we observe that our pass can handle multi-dimensional arrays of both primitive and aggregate data types. It can distinguish the row-oriented and colum-oriented access patterns `A[i][j]` and `A[j][i]`. In the former case, the access pattern is of the form :

{{0,+,(4 * %m)}<%for.cond>,+,4}<nw><%for.cond3>,

i.e. the accesses are row-oriented. In the latter case, the stride is larger and the accesses are column-oriented. This is even more evident from the delinearized access patterns, wherein the order of the for-loop conditions are inverted in the two cases.

Similary, in the case of the array of structures, the offset of the first and second character fields within the structure are identified in the access pattern. This is evident in the expressions {0,+,3}<nuw><nsw><%for.cond> and {1,+,3}<nw><%for.cond>. We have successfully tested our compiler pass on more complex access patterns. Overall, we can effectively analyze the loads and stores associated with the critical data structures using the pass.

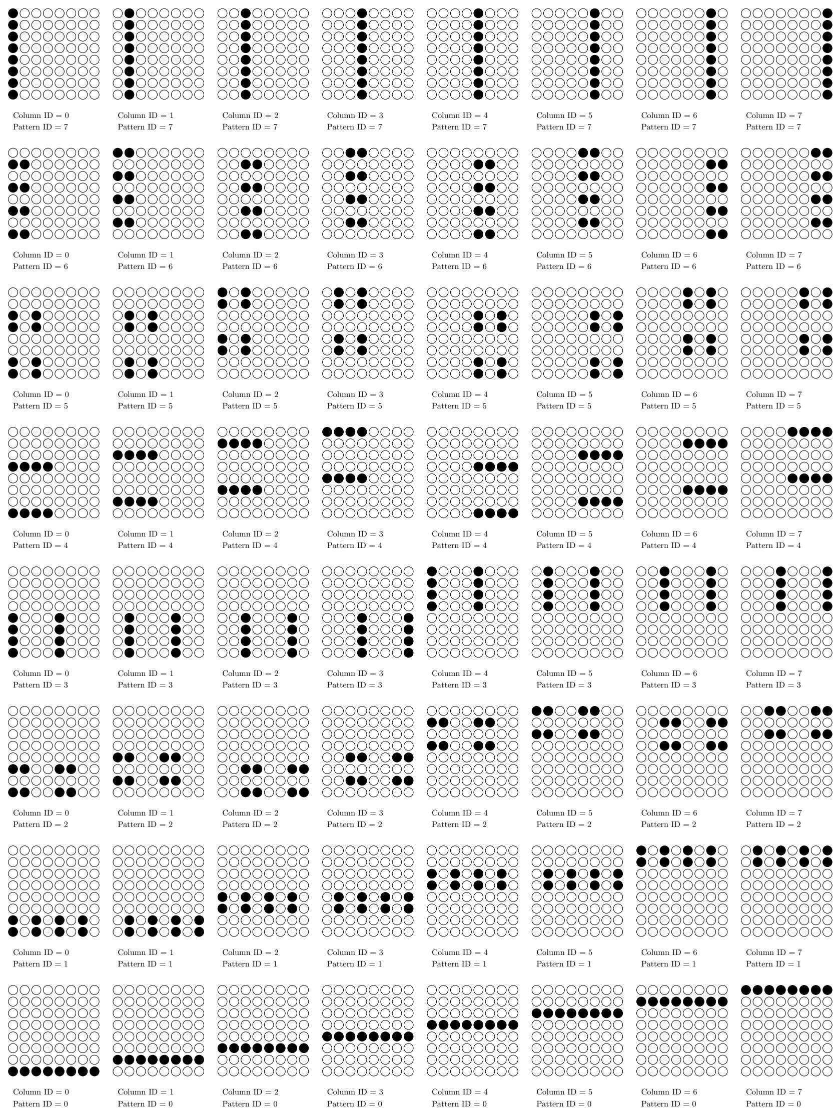

Column ID = 0 Pattern ID = 7
Column ID = 1 Pattern ID = 7
Column ID = 2 Pattern ID = 7
Column ID = 3 Pattern ID = 7
Column ID = 4 Pattern ID = 7
Column ID = 5 Pattern ID = 7
Column ID = 6 Pattern ID = 7
Column ID = 7 Pattern ID = 7
Column ID = 0 Pattern ID = 6
Column ID = 1 Pattern ID = 6
Column ID = 2 Pattern ID = 6
Column ID = 3 Pattern ID = 6
Column ID = 4 Pattern ID = 6
Column ID = 5 Pattern ID = 6
Column ID = 6 Pattern ID = 6
Column ID = 7 Pattern ID = 6
Column ID = 0 Pattern ID = 5
Column ID = 1 Pattern ID = 5
Column ID = 2 Pattern ID = 5
Column ID = 3 Pattern ID = 5
Column ID = 4 Pattern ID = 5
Column ID = 5 Pattern ID = 5
Column ID = 6 Pattern ID = 5
Column ID = 7 Pattern ID = 5
Column ID = 0 Pattern ID = 4
Column ID = 1 Pattern ID = 4
Column ID = 2 Pattern ID = 4
Column ID = 3 Pattern ID = 4
Column ID = 4 Pattern ID = 4
Column ID = 5 Pattern ID = 4
Column ID = 6 Pattern ID = 4
Column ID = 7 Pattern ID = 4
Column ID = 0 Pattern ID = 3
Column ID = 1 Pattern ID = 3
Column ID = 2 Pattern ID = 3
Column ID = 3 Pattern ID = 3
Column ID = 4 Pattern ID = 3
Column ID = 5 Pattern ID = 3
Column ID = 6 Pattern ID = 3
Column ID = 7 Pattern ID = 3
Column ID = 0 Pattern ID = 2
Column ID = 1 Pattern ID = 2
Column ID = 2 Pattern ID = 2
Column ID = 3 Pattern ID = 2
Column ID = 4 Pattern ID = 2
Column ID = 5 Pattern ID = 2
Column ID = 6 Pattern ID = 2
Column ID = 7 Pattern ID = 2
Column ID = 0 Pattern ID = 1
Column ID = 1 Pattern ID = 1
Column ID = 2 Pattern ID = 1
Column ID = 3 Pattern ID = 1
Column ID = 4 Pattern ID = 1
Column ID = 5 Pattern ID = 1
Column ID = 6 Pattern ID = 1
Column ID = 7 Pattern ID = 1
Column ID = 0 Pattern ID = 0
Column ID = 1 Pattern ID = 0
Column ID = 2 Pattern ID = 0
Column ID = 3 Pattern ID = 0
Column ID = 4 Pattern ID = 0
Column ID = 5 Pattern ID = 0
Column ID = 6 Pattern ID = 0
Column ID = 7 Pattern ID = 0

Figure 3: List of possible patterns for a 8×8 matrix with a given shuffling function. The column id refers to the cacheline being accessed. The elements of the cacheline that are highlighted are returned to the application.

3.2 DRAM Access Cost Model

We now describe the cost model we developed to help us map the application's access patterns to the relevant DRAM access primitive. To begin with, consider the list of possible patterns for 8×8 toy matrix with a given shuffling function, shown in fig. 3. We observe that there are 8 different DRAM access patterns. In each case, the *column id* refers to the cacheline being accessed, while the *pattern id* is the auxiliary information that we can supply using the novel DRAM access primitives.

The elements of the cacheline that are highlighted will be returned to the application that is requesting the desired cacheline and pattern id. We note that using this tile to DRAM row mapping, we can access both the first row and first column in the matrix in a single DRAM read operation.

3.2.1 Simple Mapping Function

Based on this observation, we started with a simple mapping function. In the delinearized access pattern, if the innermost loop is the outer condition, then clearly it is column-oriented and will benefit from pattern id 7. On the other hand, if the innermost loop is the inner condition of the delinearized access pattern, then it will benefit from pattern id 0, the default pattern in current hardware systems.

Although this works well for relatively complex workloads comprising of both row-oriented and column-oriented accesses to different data structures, this does not generalize to more complicated access patterns. We therefore refined the cost model to take into account other access patterns ranging from 1 through 6.

3.2.2 Basic Cost Model

In this model, for a given data structure, we first analyse all the cell elements accessed by the set of all patterns in the loop. This is done using the analysis described in previous subsection. We then compute the difference between that tile and each of the available DRAM access patterns. For instance, say the hybrid access pattern for the data structure looks like this :

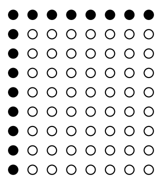


Figure 4: Sample hybrid application access pattern.

The difference between the hybrid access pattern and pattern id 0, column id 0 is 6. Similarly, the difference between the hybrid access pattern and pattern id 7, column id 0 is also 6. Either of these DRAM access primitives will be a good fit for the hybrid access pattern. In general, for two tiles $\mathcal{T}_a$ and $\mathcal{T}_b$, each of dimensions $n \times m$, is given by :

$$\text{Basic Cost}(\mathcal{T}_a, \mathcal{T}_b) = \sum_{i=1}^{n} \sum_{j=1}^{m} |\mathcal{T}_a[i][j] - \mathcal{T}_b[i][j]|$$

The lower the cost, the better the fit between the access primitive and the hybrid access pattern. Although, this simple cost model can handle more complicated access patterns, including those resembling DRAM access patterns 1 through 6, we realized that the model needs to be refined to consider cacheline data alignment.

3.2.3 Data Alignment Based Cost Model

The key observation is that when the first cell in a given DRAM row is fetched, the remaining cells in that row are also fetched into the row buffer. Therefore, not all cells in the tile are equally important. We can refine the cost model to handle data alignment in this manner.

For the tiles $\mathcal{T}_a$ and $\mathcal{T}_b$, each of dimensions $n \times m$, we first construct corresponding binary vectors $\mathcal{V}_a$ and $\mathcal{V}_b$, each of length n. For each row in the two tiles, if the sum of the elements in the row is non-zero, then the corresponding vector element is set to one, i.e.

$$\mathcal{V}_a[i] = \begin{cases} 1, & \text{if } \sum_{j=1}^{m} \mathcal{T}_a[i][j] > 0 \\ 0, & \text{otherwise} \end{cases}$$

Now, we can define the cost as :

$$\text{Data Alignment Cost}(\mathcal{T}_a, \mathcal{T}_b) = \sum_{i=1}^{n} |\mathcal{V}_a[i][j] - \mathcal{V}_b[i][j]|$$

This refined model works well for cacheline alignment. However, we will need to resolve ties between access primitives that have the same data aligment cost, but may not be equally good fits for the hybrid access pattern. We therefore take both the basic cost and data alignment cost into consideration to figure out the ideal access primitive for a given hybrid access pattern. This is defined as :

$$\begin{aligned} \text{Weighted DRAM Access Cost}(\mathcal{T}_a, \mathcal{T}_b) = {} & \alpha \times \text{Data Alignment Cost}(\mathcal{T}_a, \mathcal{T}_b) + \\ & (1 - \alpha) \times \text{Basic Cost}(\mathcal{T}_a, \mathcal{T}_b) \end{aligned}$$

Now, we can use the weighted DRAM access cost model to find the optimal DRAM access primitive for a given hybrid application access pattern. Thus, to summarize, we first analyze the loads and stores associated with the critical data structures to determine the application's access patterns and then use the cost model to map that access pattern to make use of the relevant DRAM access primitive.

4 Experimental Setup

Compiler Pass: We implemented our compiler pass in the LLVM 3.5 framework. The pass analyses the application's data structure access patterns with the help of scalar evolution-related (SCEV) functions [8]. It then uses the cost model to find the optimal DRAM access primitive among the ones shown in fig. 3 for the given hybrid data access pattern.

Gem5: Gem5 simulator is a configurable simulation framework that can be used to emulate the novel DRAM access primitives. We used a modified version of Gem5, that takes in *pattern id* in the unused address bits from the prefetch instruction to emulate the access pattern. The access function is shown below for clarity :

Listing 3: Gem5 DRAM Access Primitive Simulation

```
#define PATTERN_ROW 0
#define PATTERN_COL 7

static __attribute__((aligned(512))) long table[];

// ACCESS PRIMITIVES
static long gather(long *table, int index, int pattern) {
    char *p = (char*)&table[index] + pattern;
    long result;
    // PREFETCH INSTRUCTION
    asm("prefetch %1\n" : "=a"(result) : "m"(*p) : );
    return result;
}

int main() {

        // ROW-ORIENTED ACCESS
        for (int i = 0; i < 16; i++) {
                printf("table[%2d] by tuple: %016lx\n", i,
                    gather(table, i, PATTERN_ROW));
        }

        // COLUMN-ORIENTED ACCESS
        for (int i = 0; i < 16; i++) {
                printf("table[%2d] by field: %016lx\n", i,
                    gather(table, i, PATTERN_COL));
        }

}
```

The changes in Gem5 are being made in tandem with this work. Due to functional limitations of the current prototype, we were not able to obtain the statistics required for evaluating our benchmarks. We therefore decided to estimate the performance benefits directly on real hardware using a matrix library by emulating the different DRAM access primitives. The specifications of the system that we used in our evaluation is shown in table 1. The source code of this project is available here [5].

CPU	Intel Xeon CPU E5-2420 v2 @ 2.20GHz
Number of cores	6
Threads per core	2
L1d cache	32 KB
L1i cache	32 KB
L2 cache	256 KB
L3 cache	15 MB

Table 1: System Specification – The hardware specification of the system used for evaluation

5 Experimental Evaluation

We evaluate potential performance impact of the novel DRAM access primitives by comparing the time taken to complete the *same task* using different *DRAM patterns ids* in our custom matrix library. The task involves performing a long series of reads and writes in a wide single-dimensional array at the *tile-level granularity*. We have a knob that allows us to adjust the read-write ratio. We then compare the performance obtained using different access patterns in the system after normalizing them with respect to the default pattern id 0.

A high-level overview of the benchmark looks like this :

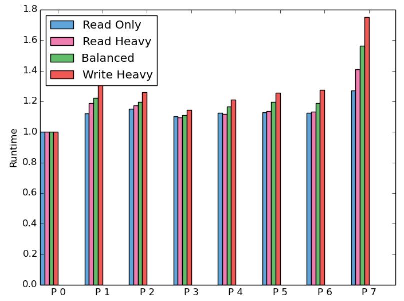
Listing 4: Benchmark

```
void Test(void * matrix, int pattern_id, int rd_wr_ratio, int
    scale) {
    StartTimer();

    for(k = 0 ; k < repeat; k++) {
        AccessMatrix(mat, pattern_id, rd_wr_ratio, scale);
    }

    StopTimer();
}
```

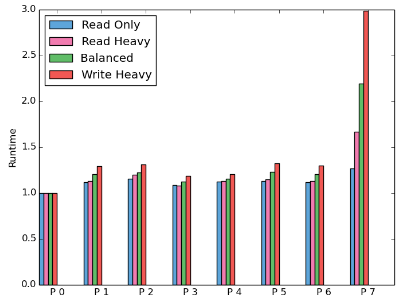
We vary the following parameters in our experiments :

- **Pattern Id:** We implemented all the 8 access patterns depicted in fig. 3.
- **Workload:** We consider four workloads (Read only, Read heavy, Balanced, and Write Heavy) with the corresponding read-write ratio being 0, 0.25, 0.5, and 0.75 respectively. The *AccessMatrix* function reads and writes cells in different tiles of the matrix using the given pattern id, based on the read-write ratio in the generated workload.
- **Tile size:** This is the dimension of the square pattern matrix. We considered the following values - 64, 128, 256, and 512.

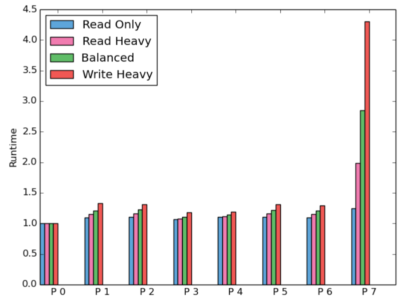
We summarize our results in fig. 5, wherein we report the normalized runtime with respect to the baseline row-major pattern *P 0*. Among all DRAM access primitives, we expected that the columnar access pattern *P 7* would perform the worst for our task, as we designed the task to be ideally suited for the baseline DRAM access primitive.



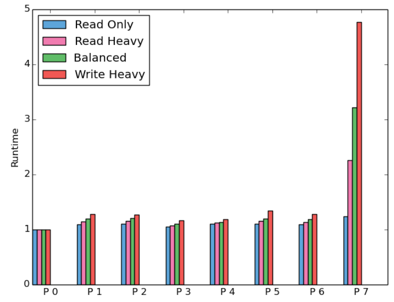
(a) 64×64 tile



(b) 128×128 tiles



(c) 256×256 tile



(d) 512×512 tiles

Figure 5: Performance impact of DRAM access primitives – Evaluation of the normalized runtime incurred while running the same task using different DRAM access primitives. We consider four tasks (Read only, Read heavy, Balanced, and Write Heavy) and four tile dimensions.

The main takeaway is that the choice of DRAM access pattern has a strong performance impact on the workload. In general, we observed that the other DRAM access primitives performed worser than the baseline access pattern on the task, to different extents. This is in line with our expectation, as the task and the current memory architecture is optimized for the default row-major DRAM access pattern. We observed that the other access patterns incur significantly higher L1 data cache misses and this increases the normalized runtime of the workload.

Furthermore, we note that the performance degradation is more pronounced in case of the write-intensive workloads (Balanced and Write-Heavy). We attribute this to the fact that write operations will need to fetch the data and also write back the modifed contents back to memory, unlike the read operations. This increases the number of row-buffer conflicts and the sensitivity to the choice of the DRAM access pattern primitive.

Finally, we examined the impact of the tile size. We observed that the effects of access primitives are more significant with larger tiles. This effect is prominently visible particulary for pattern id 7, wherein the dimension of the tile is directly related to number of cache misses incurred. Overall, we expect that depending on the hybrid application access pattern, the choice of DRAM access primitive will have a significant performance impact. Hence, there is a significant opportunity for the compiler to optimize this choice using our weighted DRAM access cost model, on future machines wherein we have these novel primitives.

6 Surprises and Lessons Learned

To begin with, we were pleasantly surprised by the support in LLVM for analysing scalar evolution (SCEV) in loops. This helped in automatically identifying the application pattern (i.e. the stride and the offset). Initially, we were considering that the programmer might need to annotate this information as well. However, we were able to obtain this information without any annotations and we could also generalize this to multi-dimensional arrays of both primitive data types and structures. This should cover a significant class of programs wherein we would like the compiler to automatically transform the program to make use of the novel DRAM access primitives.

We were initially not sure about how to go about with the performance evaluation as these DRAM primitives are not available on any machine. We tried to make use of the Gem5 prototype to do a full system evaluation. However, this proved to be too challenging as the prototype needed more work that was beyond the scope of this project. Even though this turned out to not be directly useful, we still got to run some interesting microbenchmarks on the full system simulator and got to appreciate the complexity of the full system simulator.

An important lesson we learned is that the DRAM access pattern has a very significant impact on the application performance. We expected to observe some performance impact, but we did not anticipate that we would be able to obtain 2-5× speed-up out-of-the-box for the same task by simply altering the DRAM access pattern. Another interesting phase in the project was refining the cost model over time, from a simple mapping function to one that takes all the available DRAM access primitives as well as the data alignment into consideration.

7 Conclusions and Future Work

We developed a compiler pass that allows us to automatically identify the hybrid application access pattern and optimize the choice of the underlying DRAM access primitives using a cost model. We examined the potential benefits of this optimization in our evaluation, wherein we varied the pattern type, workload, and tile size to observe the impact on runtime performance. Our evaluation showed that choosing the right DRAM access primitive for the required task using a cost model can provide 2-5$\times$ speed-up.

In this work, we restricted our analysis to the canonical DRAM access patterns and used a synthetic low-level matrix-based library. We think that an extensive evaluation on more realistic workloads will provide a better understanding of the performance impact of these primitives and raise other interesting problems. After the prototype in the Gem5 simulator is fully functional, we can do a more accurate cache performance analysis. For instance, it will interesting to see the impact of these primitives on the traditional cache coherence protocols in case of write operations.

8 Distribution of Total Credit

Joy: 37.5%, Junwoo: 35%, and Anuj: 27.5%.

References

[1] Abadi, D. J., Madden, S. R., and Hachem, N. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 967–980.

[2] Ailamaki, A., DeWitt, D. J., and Hill, M. D. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal 11*, 3 (Nov. 2002), 198–215.

[3] Boncz, P., Zukowski, M., and Nes, N. Monetdb/x100: Hyper-pipelining query execution. In *In CIDR* (2005).

[4] Copeland, G. P., and Khoshafian, S. N. A decomposition storage model, 1985.

[5] Github. Loop Memory Access Analysis. `https://github.com/jarulraj/llvm/tree/master/project`.

[6] Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., and Madden, S. HYRISE: a main memory hybrid storage engine. *Proc. VLDB Endow. 4*, 2 (2010), 105–116.

[7] Jeong, M. K., Yoon, D. H., Sunwoo, D., Sullivan, M., Lee, I., and Erez, M. Balancing dram locality and parallelism in shared memory cmp systems. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on* (Feb 2012), pp. 1–12.

[8] LLVM. LLVM SCEV class reference. `http://llvm.org/docs/doxygen/html/classllvm_1_1SCEV.html`.

[9] Raman, V., Attaluri, G., Barber, R., Chainani, N., Kalmuk, D., KulandaiSamy, V., Leenstra, J., Lightstone, S., Liu, S., Lohman, G. M., Malkemus, T., Mueller, R., Pandis, I., Schiefer, B., Sharpe, D., Sidle, R., Storm, A., and Zhang, L. Db2 with blu acceleration: So much more than just a column store. *Proc. VLDB Endow. 6*, 11, 1080–1091.

[10] Sudan, K., Chatterjee, N., Nellans, D., Awasthi, M., Balasubramonian, R., and Davis, A. Micro-pages: Increasing dram efficiency with locality-aware data placement. *SIGARCH Comput. Archit. News 38*, 1 (Mar. 2010), 219–230.

[11] Zhang, Z., Zhu, Z., and Zhang, X. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture* (New York, NY, USA, 2000), MICRO 33, ACM, pp. 32–41.

[12] Zukowski, M., Boncz, P. A., Nes, N., and Hman, S. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull. 28*, 2 (2005), 17–22.