Virtual Views

Kevin Chang, Vivek Seshadri
kevincha@cmu.edu, vseshadr@cs.cmu.edu

1 Introduction

In modern systems, memory performance is highly critical for application performance. Modern processors use multiple levels of caches to exploit both temporal and spatial locality present in different access patterns of applications. In this work, we focus our attention on the class of applications which access large data structures using multiple different access patterns – e.g., a matrix application that access a matrix both in row-major order and column major order, or a database application in which certain queries access most fields of a few records whereas other queries touch very few fields of many records.

1.1 The Problem

In such applications that access large data structures with multiple access patterns, when the data structure is stored in only one format in physical memory can lead to inefficiencies in the memory hierarchy for some of the access patterns. For example, consider a matrix application that accesses a matrix both in row-major order and column-major order. If the matrix is stored only in the row-major order, like in most systems, then the column major access pattern will have zero spatial locality. However, almost all components of the memory subsystem – i.e. caches, DRAM, prefetchers – assume that applications have some spatial locality. As a result, the column major access pattern will lead to unwanted data movement between memory components. In addition, such access patterns with low spatial locality can also increase the working set of an application in terms of the number of cache blocks. This can result in thrashing when there is not enough space in the cache to hold the working set. On the other hand, if the matrix is stored only in the column-major order, then the row-major access pattern will suffer from the above mentioned problems. In this project, we examine an approach to address this problem that can ensure efficient utilization of the memory hierarchy for all access patterns of such applications.

1.2 Virtual Views: Our Approach

Our idea is to maintain multiple versions of a data structure in physical memory, each catering to one or more of the different access patterns that access the data structure. We call each one of these versions *a Virtual View* of the data structure. We leverage the use of the compiler to ensure that the different views of the data structure are kept consistent. Depending on the nature of an access pattern, the application can choose a specific view of the data structure that provides the best performance. For example, for the matrix application, our technique will store both the row-major view and the column-major view of the matrix. When accessing the matrix in the row-major order, it will use the row-major view and when accessing the matrix in the column-major order, it will use the column-major view.

To keep the multiple views of a data structure consistent, whenever a piece of data in one view is modified, the updated value should be propagated to the corresponding elements of the other views. This requires the compiler to generate additional writes. We explore two different approaches to decide when to perform these additional writes: 1) *eager write*, 2) *lazy regenerate*. We describe these two approaches in more detail in Section 3.

1.3 Related Work

Prior work had addressed this inefficiency problem in the memory hierarchy using other approaches [5, 8]. Active pages [5] proposes a design to export certain computations to the memory controller. The Impulse project [8] proposes a new memory controller design wherein the memory controller ensures that only useful data is sent across the channel to the processor. The application specifies which pieces of data within a data structure are useful and this information is stored in the memory controller in the form of mapping tables. When the application generates an access to the data structure (using a newly assigned name), the memory controller gathers the useful data from the main memory and sends it to the processor. This ensures efficient memory bandwidth utilization and cache utilization. One major drawback of the Impulse approach is that the application needs to flush the cache before and after creating the new mappings to ensure correctness. Chilimbi et al. [1] propose to coallocate data structures that appear in hot paths of execution to improve the spatial locality along those paths. However, the proposed scheme will work not work if there are multiple hot paths with different access patterns.

1.4 Contributions

- We explore a new approach, Virtual Views, to address the memory inefficiency problem, where an application accesses a single data structure with multiple different access patterns.
- We explore different design options that fit our proposed framework. Specifically, we propose two different strategies to maintain consistency across virtual views.
- We partially implement our proposed optimizations using the LLVM compiler framework.
- We evaluate the performance effect of the our proposed schemes using a microbenchmark that represents a database application.

2 Access Patterns & Locality

As mentioned in Section 1, we address the memory inefficiency problem using an approach, which we call *Virtual Views*. In this section, we define the problem more formally before we describe our proposed approach in detail.

An *access pattern*, with respect to a data structure, refers to an ordered sequence of elements (words) of the data structure that are accessed by the application. Most access patterns exhibit locality of reference, which is generally classified into two types: temporal locality and spatial locality. Temporal locality refers to the reuse of a specific piece of data within a small time duration. Spatial locality refers to use of data co-located in physical memory within a small time duration. Figure 1 shows four different access patterns to an array of elements, A. The access patterns contain either, both or none of temporal or spatial locality.

A[0], A[100], A[0], A[100], A[0], A[100]

(a) Temporal Locality

A[0], A[1], A[2], A[3], A[4], A[5]

(b) Spatial Locality

A[0], A[1], A[2], A[0], A[1], A[2]

(c) Temporal and Spatial Locality

A[0], A[104], A[35], A[220], A[75], A[151]

(d) No Locality

Figure 1: Access patterns to an array, A

There is a plethora of prior work which has focused on exploiting temporal locality present in access patterns by modifying the replacement policies used for caches and DRAM (e.g., [2, 4, 6, 7]). However, the spatial locality in access patterns is tightly coupled with the order in which data is stored in physical memory. If the two do not match, then the access pattern will not exhibit good spatial locality. Figure 2 shows a scenario where there are two different access patterns to a matrix that is stored in row-major order, i.e., subsequent elements of a row are stored in adjacent locations in physical memory. While one of them, the row-major access pattern, exhibits very good spatial locality, the column-major access pattern exhibits zero spatial locality.

A[0][0], A[0][1], A[0][2], A[0][3], A[0][4], A[0][5]

(a) Row-major access: Exhibits spatial locality

A[0][0], A[1][0], A[2][0], A[3][0], A[4][0], A[5][0]

(b) Column-major access: Does not exhibit spatial locality

Figure 2: Two access patterns to a matrix stored in row-major order

Consider an application that accesses a matrix using both the row-major and column-major access patterns. While the memory hierarchy (caches and DRAM) can exploit the spatial locality presented by the row-major access pattern, they will be completely inefficient for the column-major access pattern for two reasons. First, since only one element within a cacheline will likely be used, the column-major access pattern can lead to cache underutilization. Second, to ensure high density, DRAM's employ large row-buffers (8KB/row). On every cacheline access, a full row of data is accessed and the required cacheline is sent to the processor. If subsequent accesses go to the same row, they are served quickly off the row-buffer. However, if there is no spatial locality and subsequent accesses go to a different row, they incur much longer latency as the data must first be fetched from the DRAM cells. Prior work has proposed some techniques improve spatial locality in access patterns [1, 8], they [8] either are not amenable to modern processors which have on-chip memory controllers or they assume that the data structures are accessed with a fixed access pattern. In this work, we explore an alternate approach to improve spatial locality of multiple access patterns to a data structure.

3 Virtual Views

The key idea behind our approach is to store multiple versions of a data structure in physical memory, each matching a specific access pattern to the data structure. We call each one of these versions a *Virtual View* of the data structure. We leverage the use of compilers to keep the different views of the data structure consistent.

For example, for our matrix application that accesses a matrix both in row-major and column-major order, our approach stores two views of the same matrix: one storing the matrix in the row-major order (A) and another storing the matrix in the column-major order (B). When the application needs to access the matrix using the row-major access pattern, it uses the view A, and for the column-major access pattern, it uses the view B. The main challenge involves the compiler keeping the two views A and B consistent. This requires the compiler to identify write operations to the views and generate additional writes to the corresponding elements in the other views. We call the generation of these additional writes as *write amplification.*

Our approach of using Virtual Views has two components. First, it requires the application or the compiler to identify the opportunity for using Virtual Views. Although exploring the latter was part of this project's goal, we resort to the former, where the application explicitly specifies the opportunity for exploiting Virtual Views. Automatic Virtual View generation requires sophisticated static analysis (or profiling) of the application to identify opportunities for creating Virtual Views. Due to lack of time, we leave it as part of future work.

We explore two ways of implementing *write amplification* (described in Section 4): 1) eager write, and 2) lazy regeneration. The eager write approach eagerly generates write operations to all the views when it sees a write operation to one of the views. This is more like the update-based coherence protocol where the data is propagated immediately to multiple views. On the other hand, on a write to a view, the lazy regenerate approach marks all other views as stale and regenerates them just before they are read. While the eager write approach may work better when accesses to different views are interleaved at a fine granularity, when there are multiple writes to the same location, it might often be sufficient to generate an additional store for the last write. On the other hand, lazy regenerate will work well when accesses to different views of the data structure are interleaved at a courser granularity. However, the lazy regenerate approach will not work well when the fraction of writes is low.

4 Implementation

In this project, we require compiler support to 1) identify Virtual Views by analyzing the applications that explicitly declare uses of them, and 2) maintains consistency between data structures and their views through write amplification. To accomplish these tasks, we use the LLVM 3.0 compiler framework [3] to compile, analyze, and transform applications for our project.

4.1 Virtual View Identification

Since the goal of our project is to explore the potential benefits of using Virtual Views, we modify all our applications to explicitly declare and initialize Virtual Views by calling a pragma (*VIRTUAL_VIEW*) instead of having the compiler to automatically generate Virtual Views. As a result, our LLVM analysis simply examines the arguments of each pragma to identify the Virtual View and its corresponding data structure. We keep track of each linkage in order for us to perform correct write amplification.

4.2 Virtual View Write Amplification

We wrote two LLVM passes that transform the applications to maintain consistency across data structures and their views. Specifically, we implemented two variants of write amplification: 1) *eager write* and 2) *lazy regenerate* as described in Section 3.

The eager write pass first runs the Virtual View identification pass to find all Virtual Views and their source data structures. After that, the pass sequentially runs through the application's intermediate representation (IR) to find all store instructions. Once a pass verifies that a store instruction's pointer points to an element of a data structure that has a Virtual View, it *eagerly* generates a new store instruction with the same value, but a different element pointer, pointing to the corresponding element in the other view. Note that we assume there is no pointer aliasing issue in our target applications.

The lazy regenerate approach we implemented regenerates an entire stale view from the source data structure. Similar to the eager write pass, it has to identify all Virtual Views and their source data structures. However, when it finds a store instruction that writes to a source data structure, it simply marks the data structure as dirty instead of duplicating the store instruction. The pass inserts a re-generation function call (provided by the applications) only when there is a read to the view that has the dirty bit set for its source data structure. This can reduce the write overhead of using Virtual Views when there can be a lot of redundant writes to the data structure.

5 Benefits and Limitations

The motivation of our project is to tackle the problem of different access patterns that result in poor locality due to way data is laid out in the memory. In this work, we propose to use Virtual Views to store data in multiple different layouts in an attempt to solve this problem. We have described the mechanism and its implementation in the previous sections. We now qualitatively discuss the benefits of using Virtual View.

Using Virtual View provides several benefits. First, by having multiple data layouts, Virtual Views retain spatial locality for various access streams that access the elements of the data structure in different orders. Hence, using Virtual View can significantly improve cache and memory performance with much lower cache miss rate and memory bandwidth consumption. Second, Virtual View provides automatic data consistency across views and sources through compiler support. Additionally, compilers can also provide automatic Virtual View creation along with automatic read transformation that selects the highest locality data layout to read from. As a result, the support of Virtual View can be completely transparent to the users.

On the other hand, there are also some limitations or weaknesses of Virtual View. First, it requires additional memory storage to hold multiple copies of data. This could be a major concern for programs that have big data and want to create multiple views for each data structure. Second, write amplification to ensure data consistency can become a significant bottleneck for performance when there is an abundant of writes, increasing the memory bandwidth consumption. This is especially true for *eager write* that duplicates every single write. In order to mitigate this, we proposed using *lazy regenerate* that delays all writes to a view until a read needs to be performed on that view. Doing so eliminates redundant writes to the same memory location, and thus reduces pressure on the cache and the memory. Given the tradeoff of using Virtual View, application developers can decide to use Virtual View based on their needs. If they care more about performance and memory capacity is not a major issue, Virtual View can significantly improve application performance with acceptable overhead.

6 Methodology

This section describes the system and microbenchmark we use to experimentally evaluate Virtual View. We present quantitative results of our experiments in the next section.

6.1 System

We use a real system that has a reasonable cache and memory capacity to evaluate Virtual View. Table 1 shows the detailed configuration of our system. The system has the hardware prefetching enabled. We will evaluate Virtual View without prefetching as part of the future work.

Processor	Intel Xeon 3.00, X5472 GHz
L1 Cache Size	32KB, 8-way associative
L3 Cache Size	12MB
Cacheline Size	64B
Main Memory	8GB, DDR3, 2 channels
OS Kernel	Linux kernel 2.6.38-11
OS	Ubuntu 11.04 64-bit (x86_64)

Table 1: System Configuration

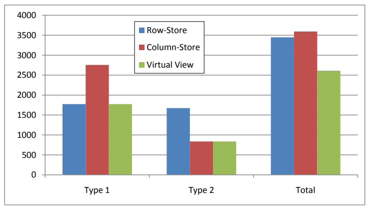
6.2 Database Microbenchmark

The goal of this project is to improve the performance of applications that access certain data structures with multiple different access patterns. To this end, we created a microbenchmark which represents a database application that runs two types of queries on a table. Type 1 query: a short transaction that accesses all fields of a particular record. Type 2 query: a long running operation that touches only a fraction of the fields of all records. Type 1 queries benefit from a row-oriented organization of the table whereas type 2 queries benefit from a column-oriented organization of the table. In addition, the microbenchmark also models updates to the database which can be controlled using command line parameters. The benchmark runs multiple queries of type 1 and type 2 in batches and the fraction of those queries that modify the records can be varied. To evaluate Virtual View, we created both row- and column-major order data structures. The benchmark chooses which data structure to use based on the access pattern. Furthermore, we also implemented both *eager write* and *lazy regenerate* in order to compare their performance.

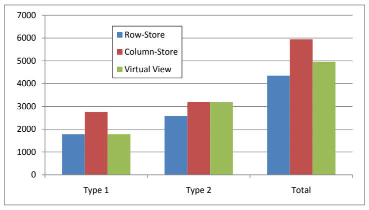
7 Results and Analysis

7.1 Read-only Query Performance

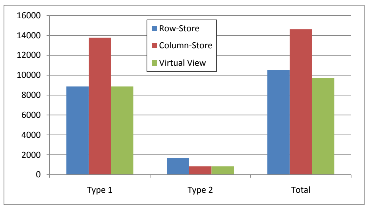
Figure 3 shows the performance improvement of using Virtual Views for different query configurations – i.e., the fraction of records touched by type 1 queries and fraction of fields touched by type 2 queries. In this experiment, all queries are read-only. Hence, there is no additional overhead due to write amplification in these results. Several observation are in order. First, as expected, the row-store organization significantly outperforms the column-store organization for type 1 queries and the column-store outperforms the row-store for type 2 queries for most cases. Second, in all cases, virtual views obtains the performance of the row-store for type 1 queries and that of the column-store for type 2 queries. As a result, it provides the best performance overall. Third, when the type 2 queries access all fields, they essentially access all the data in the database. In this special case, a row-store organization is better as it offers better spatial locality than the column-store organization. Consequently, it outperforms the column-store organization even for type 2 queries. When using virtual views, type 2 queries are served by the column-store. Hence, as expected, the performance of



(a) Frac. of records: 0.2, Frac. of fields: 0.2



(b) Frac. of records: 0.2, Frac. of fields: 1



(c) Frac. of records: 1, Frac. of fields: 0.2

Figure 3: Performance comparison of Virtual Views with a row-store only or column-store only organization. All queries in this experiment are read-only.

virtual views is similar to that of a column-store. Based on these results, we conclude that virtual views can be a good technique to improve performance of applications that have multiple access patterns to a given data structure when the fraction of writes to the data structure is close to zero.

7.2 Effect of Writes

Figure 4 plots the effect of varying the write probability for the type 1 queries. As the figure shows, even a small write probability (0.05) degrades the performance of different mechanisms, except the column-store organization. This is because, all our queries write to only one field and as a result, the column-store organization leads to only a small portion of the working set that is written to. The performance gap between the eager write approach and lazy regenerate approach is not significant in this experiment.

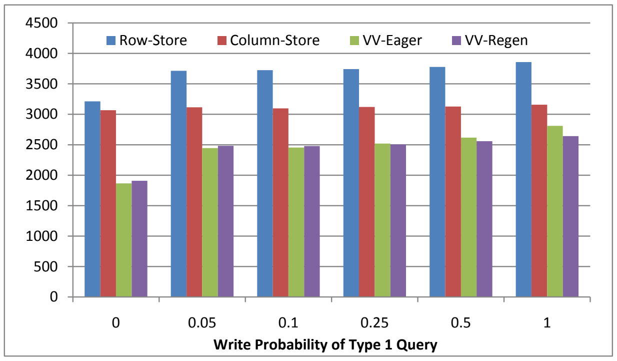


Figure 4: Effect of varying the write probability of type 1 queries

Figure 5 plots the effect of varying the write probability for the type 2 queries. In this study, if a query is chosen to update the records, it updates all the records. Therefore, when multiple queries choose to update the records, there will be a lot of redundant writes to the database. This is evident from the steep degradation in performance of the eager write technique. As the probability reaches 1, when all queries update all records, the performance of the eager write technique is as bad as the row-store organization. However, the lazy regenerate technique amortizes the cost of writes by performing it only once after all queries are completed. As a result, it experiences much less performance degradation compared to the eager write technique.

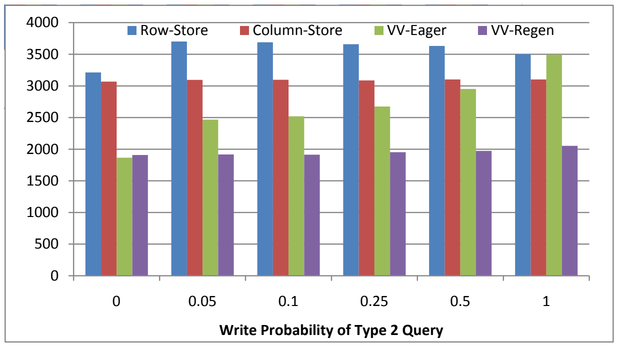


Figure 5: Effect of varying the write probability of type 2 queries

8 Conclusion

In this project, we propose Virtual View, a new approach to address the spatial locality problem caused by disparate access patterns issued by applications. The key idea of Virtual View is to maintain multiple layouts of a data structure in physical memory in order to retain high spatial locality for multiple commonly used access patterns (e.g., row-major order). To maintain data consistency across views and their source data structures, we propose *write amplification*, which generate additional writes to views when an element in the source data structure is updated. We describe two approaches of implementing write amplification: *eager write* and *lazy regeneration*. Eager write simply generate new writes to all the views when there a write is issued to their source. Since eager write can potentially consume significant write bandwidth, we use lazy regeneration to mitigate this overhead. This approach only performs writes to Virtual Views when there is a read from any view and the source is dirty. Virtual View provides two key benefits. First, using Virtual View improves application performance with reduced cache miss rate for applications that have multiple access patterns to the same data structure. Second, compilers provide support to automatic Virtual View creation and data consistency.

Our experimental evaluations demonstrate the potential of Virtual View for improving spatial locality, which leads to application performance gain. In addition, we also compare the performances of write amplification variants, and we show that *lazy regeneration* performance gain becomes greater as the frequency of write increases. We conclude that Virtual View can be an effective approach to exploit cache locality for applications that have the presence of multiple disparate access patterns.

References

[1] Trishul M. Chilimbi and Ran Shaham. Cache-conscious coallocation of hot data streams. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, 2006.

[2] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction. In *ISCA-37*, 2010.

[3] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[4] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST-2*, 2003.

[5] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: a computation model for intelligent memory. In *Proceedings of the 25th annual international symposium on Computer architecture*, ISCA '98, 1998.

[6] Moinuddin K. Qureshi, A. Jaleel, Y.N. Patt, S.C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA-34*, 2007.

[7] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling ways and associativity. In *MICRO-43*, 2010.

[8] Lixin Zhang, Zhen Fang, Mide Parker, Binu K. Mathew, Lambert Schaelicke, John B. Carter, Wilson C. Hsieh, and Sally A. McKee. The impulse memory controller. *IEEE Trans. Comput.*, 50(11):1117–1132, November 2001.