The Hardness of Data Packing

Rahman Lavaee

Department of Computer Science,
University of Rochester,
Rochester, NY, USA
rlavaee@cs.rochester.edu

Abstract

A program can benefit from improved cache block utilization when contemporaneously accessed data elements are placed in the same memory block. This can reduce the program's memory block working set and thereby, reduce the capacity miss rate. We formally define the problem of data packing for arbitrary number of blocks in the cache and packing factor (the number of data objects fitting in a cache block) and study how well the optimal solution can be approximated for two dual problems. On the one hand, we show that the cache hit maximization problem is approximable within a constant factor, for every fixed number of blocks in the cache. On the other hand, we show that unless $P = NP$, the cache miss minimization problem cannot be efficiently approximated.

Categories and Subject Descriptors D.3.4 [Processors]: Memory management (garbage collection), Optimization; F.2.2 [Nonnumerical Algorithms and Problems]: Computations on discrete structures

General Terms algorithms, performance, theory

Keywords Data packing, cache block utilization, proximity hypergraph, block reuse interval

1. Introduction

Today's caching systems store data in fixed size blocks. CPU caches typically use a block size of 64 bytes or 128 bytes, and memory and TLBs use a page size of 4 KB or larger. When a cache miss occurs, a full block of data is fetched, which includes other data that may be used before the block is evicted from cache. In other words, a cache block exploits the program's **spatial locality**. However, for the block-based design to be beneficial, the program must have strong enough spatial locality. In fact, if the program shows weak spatial locality, having a large data block may even be counter-productive. For instance, if among the total 16 words in every cache block, only 4 words are useful, then the cache will waste 75% of its capacity. Moreover, the data transfer bandwidth will be wasted by 75% when serving cache misses.

Therefore, it is important to maximally utilize a cache block while it is residing in the cache. Intuitively, one can improve the cache-block utilization by grouping data that are frequently accessed

closely together in time. To design an approach based on this intuition, and in order to guarantee the best utilization, one must address two important questions. First, how close in time do those accesses have to happen? And second, how frequent do those close accesses need to be? Of course, precisely answering these questions requires full knowledge of the cache details, including the cache size, the block size, associativity, the replacement scheme, and the access latency. Moreover, today's complex and hierarchical memory systems, which consist of different levels of caches with different parameters and details, make the problem much more challenging.

Generally, the *cache-conscious data placement (data layout)* problem is expressed as follows. Given all the details about the cache (including its data replacement policy) and the sequence of references to data objects of known sizes, find a placement of these objects in memory that minimizes the number of cache misses. Petrank and Rawitz [27] showed, by reducing from the k -colorability problem, that for every set-associative cache with at least three sets, and under any *sensible* replacement policy (as they define sensibility for replacement policies), this problem is **NP**-hard to approximate even within a liberal approximation factor. That is, they prove that for any $\epsilon > 0$ and for a reference sequence of length N , the problem cannot be approximated within a factor of $O(N^{1-\epsilon})$ in polynomial time, unless $P = NP$. They first proved this result assuming all objects have the same size (the size of a cache block) and then explained that their result remains valid for the general case.

The source of hardness in the proof of Petrank and Rawitz is conflict misses. Consequently, this result strongly opposes tackling the problem of conflict miss minimization, but does not give insight on whether we can attack capacity misses. There are several reasons why it is important to focus on capacity misses.

- First and Foremost, while full associativity may still be impractical for most hardware caches, it is popularly employed by software caches. A few examples are virtual memory systems, Memcached [13], and caching for embedded devices [20]. Besides this, a number of papers propose software management designs for hardware caches which boost their performance to the same level as or even higher than the same-size fully-associative caches [17, 29, 34].
- Second, today's architectures are moving towards higher associativity in cache, making the cache capacity an increasingly important factor. For instance, the associativity in the L3 cache of Intel processors has increased from 8-way in Pentium 4 to 20-way in Sandy Bridge. TLBs are often fully-associative, for example with 48 and 32 entries respectively on recent ARM processors Cortex-A72 [2] and Cortex-A57 [1].
- Third, a study conducted by Cantin and Hill [8] indicates that as the number of ways increases, the number of conflicts diminishes. That is, the difference between a set-associative cache and a fully-associative cache becomes vanishingly small. Part of their

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-3549-2/16/01...\$15.00
<http://dx.doi.org/10.1145/2837614.2837669>

data (shown in Table 1) reports the average miss rates for four 256 KB LRU caches with different associativities, when running the SPEC CPU2000 benchmark suite.

fully-associative	0.0163969
8-way set-associative	0.0167211
4-way set-associative	0.0167639
direct-mapped	0.0193043

Table 1. Average miss ratios for four 256 KB LRU caches with different associativities, when running the SPEC CPU2000 benchmark suite (adopted from [8]).

They also report a **compulsory** miss rate of 0.0000107867 for this benchmark suite. Subtracting this value from the miss ratio for the fully-associative cache gives the **capacity** miss rate, which equals 0.0163861133. This implies that capacity misses constitute 84.9% of total misses for the direct-mapped cache, 97.7% of total misses for the 4-way set-associative cache, and 97.9% of total misses for the 8-way set-associative cache. So we can infer that the performances of today’s highly associative caches are mostly governed by their capacity limit.

- Finally, from an algorithmic perspective, it is more general to focus on capacity misses. In fact, hierarchical memory models such as the external memory model (also known as EMM) [3] and the cache-oblivious model [14] assume a fully-associative internal memory (cache). The two models ignore cache conflicts. The external memory model gives the programmer full control over data replacement in cache, and the cache-oblivious model assumes an optimal replacement policy.

Notably, for a fully-associative cache, if all data objects are of the same size as the cache blocks and are required to be aligned at memory block boundaries, then data layout does not affect the performance. However, if data objects are smaller than the cache blocks, one can reduce the capacity miss rate by carefully packing objects into memory blocks. Interestingly, even if the object sizes are greater than, but not a multiple of the block size, one can still achieve a higher overall cache utilization by carefully aligning objects next to each other in different memory blocks.

In this paper, we make several contributions.

- We formally define the problem of **DATA PACKING** for every number of blocks in the cache, and every packing factor (the number of data objects fitting in a cache block).
- We show that for every number of cache blocks starting at 5, and every packing factor starting at 2, the cache miss minimization problem cannot be approximated in polynomial time, even within an approximation factor of $O(N^{1-\epsilon})$, where N is the length of the reference sequence. Our result applies to all *sensible* replacement policies (as defined by Petrank and Rawitz [27]).
- We show that even for 3-block and 4-block caches, a constant factor approximation algorithm would falsify the so-called *Unique Games Conjecture*.
- In contrast to the miss minimization problem, we show that with LRU replacement policy, the cache **hit maximization** problem admits constant factor approximation, though that constant is linearly related to both the number of cache blocks and the packing factor. We further show that this constant is tight with respect to the number of blocks in the cache.

2. Data Packing

In this section, we define the problem of *data packing* for fully-associative cache, and present the preliminary results about its complexity.

We define the optimization problem of *data packing* as follows.

Definition 2.1 (Data Packing). *Consider a memory subsystem that consists of an infinitely large memory, and a fully-associative cache with m blocks, a block size of B , and known replacement policy. Let T be a sequence of references to data objects of equal size B/p , for an integer p that we call the *packing factor*. Find a placement of objects in memory (a *placement scheme*) that minimizes the number of cache misses incurred by the reference sequence T .*

Since the cache is fully-associative, and the block size is a multiple of the object size, the placement scheme matters only within the memory blocks. In other words, as far as the cache performance is concerned, a *data placement scheme* can be represented by a *data packing scheme*. A data packing scheme specifies which objects are to be placed together within the same memory blocks.

Thabit [36] was the first to study this problem. For the simple case of one-block cache, he established a relation between this problem and the weighted version of the G -partitioning problem. The weighted version of G -partitioning is defined as follows.

Definition 2.2 (Weighted G -Partitioning [22]). *Given a weighted graph H and a graph G , find a partition of H into disjoint subgraphs isomorphic to G that maximizes the sum of the weights of the edges in those subgraphs.*

Kirkpatrick and Hell [22] showed that if G has more than 2 vertices, this problem is **NP-hard**.

Given a data packing instance for the **one-block cache**, Thabit constructs a **proximity graph** H over the set of data objects accessed in the reference sequence. For every two data objects a and b , the weight of the edge (a, b) in H is set equal to the number of times a and b occur next to each other in the reference sequence. Thabit argues that each data packing scheme is equivalent to a partition of H into complete graphs with p vertices (K_p). Additionally, for every data packing scheme, the number of cache misses (and analogously, the number of cache hits) incurred by the reference sequence, after applying that packing scheme, is equal to the sum of the weights of the edges whose endpoints are in different partitions (and analogously, in the same partition) in the corresponding graph partitioning solution.

With the above construction, Thabit falsely infers that the data packing problem is **NP-hard**, for any packing factor starting at 3. The acute reader notices that Thabit’s reduction is not of the right direction for establishing **NP-hardness**. Furthermore, the naïve inverse reduction cannot be applied here since it cannot map from every weighted G -partitioning instance. In fact, even for a correct inverse reduction, it would take super-polynomial time to construct an equivalent reference sequence for a given proximity graph. Thus even if such a reduction exists, it would not be a polynomial-time reduction.

Next we establish **NP-hardness** for the data packing problem by reducing from a different problem.

Theorem 2.1. *Data packing (Definition 2.1) is **NP-hard**, for every packing factor starting at 3.*

Proof. We prove this result for the one-block cache, and for any packing factor $p \geq 3$. Extending the result to the general case is straightforward. We reduce from the unweighted version of the following problem.

Definition 2.3 (Bounded-Size Graph Partitioning). *Given a weighted (or unweighted) graph G , and an integer p , find a parti-*

tion of G 's vertices into disjoint sets, each of size at most p , that minimizes the sum of the weights of (or in the unweighted version, the number of) edges running between different partitions.

This problem can easily be shown to be **NP**-hard for every p starting at 3, via a reduction from K_p -partitioning (the case where $p = 2$ is equivalent to maximum matching).

Given an instance of the unweighted bounded-size graph partitioning problem with $p \geq 3$, we construct a data packing problem instance for a one-block cache and with packing factor p . In the construction of our reference sequence, we will use the operator $\odot$ to denote the concatenation of references. We associate a data object for every node in the graph. Additionally, we introduce p dummy objects $d_1, d_2, \dots, d_p$. For every edge (a, b) in the partition problem, we append the sequence $a \odot b \odot d_p$ to the reference sequence. Finally, we append $3e + 1$ copies of the sequence $D = d_1 \odot d_2 \odot \dots \odot d_p$ to the reference sequence, where e is the total number of edges in the graph.

First, we argue that every optimal packing scheme must pack all the dummy objects together in the same memory block. Consider a data packing scheme that does not pack all the dummy objects together. Under this packing, every copy of D in the reference sequence incurs at least one miss. Thus overall, the reference sequence incurs more than $3e$ cache misses. On the other hand, packing all the dummy objects together guarantees incurring at most $3e$ cache misses. That is because all the references in the copies of D will hit in the one-block cache. Thus an optimal data packing scheme must pack all the dummy objects together.

Grouping the dummy objects together leads to cache misses for the first e references to d_p (those occurring in the sequences corresponding to edges of the graph). Additionally, for every edge (a, b) , the reference corresponding to a incurs a cache miss. Every other cache miss will correspond to an edge running between different partitions in the corresponding bounded-size partitioning solution. Thus an optimal bounded-size partition is equivalent to an optimal data packing scheme.

This completes the proof that for a one-block cache, data packing is **NP**-hard for every packing factor starting at 3. $\square$

We mention in passing, that for the case of one-block cache, it is more rational to formulate the data packing problem as a bounded-size partitioning problem rather than as a G -partitioning problem (in particular, K_p -partitioning), because the K_p -partitioning problem restricts the size of each partition to be exactly p , while the bounded-size partitioning problem merely bounds the number of vertices in each partition to be at most p . In practice, a data packing scheme is allowed to pack less than p objects in a memory block and leave a portion of that memory block vacant. In fact, it is easy to construct a data packing instance where an optimal packing scheme never packs p objects in the same block. Nevertheless, despite the restriction imposed by K_p -partitioning, one still has a way to reduce the bounded-size partitioning to this problem: given an instance of the bounded-size partitioning problem on a graph with n vertices, add $n(p - 1)$ dummy vertices to the graph and connect them to the n original vertices via $n^2(p - 1)$ zero-weight edges. A K_p -partition in the new graph would correspond to a bounded-size partition in the original graph and vice versa.

A common approach for tackling **NP**-hard optimization problems is to design *approximation algorithms*. Those are efficient algorithms that deliver reasonably good solutions. For example, constant-factor approximation algorithms deliver objective values which are within a constant factor of the optimal. In the next section, we investigate approximation algorithms for the one-block cache. Then, in section 4, we present our most important result: for every number of blocks in the cache starting at 5, the data packing problem

is **NP**-hard to approximate within any reasonable approximation factor.

3. Approximation for Single-Block Cache

We mentioned in section 2 that for the one-block cache, a data packing problem can be formulated as a bounded-size partitioning problem. The existing literature does not study the approximability of this problem. However, the bounded-size partitioning problem is itself very closely related to the (k, ν) -balanced graph partitioning problem [4], which is studied in depth. This problem is defined as follows.

Definition 3.1 ((k, ν) -balanced Graph Partitioning). *Given a weighted graph with n vertices, divide its vertices into k almost equal-size components, each of size less than $\nu \frac{n}{k}$, so that the total weight of edges between different partitions is minimized.*

When $\nu = 1$ the problem asks for a **perfectly** balanced partitioning, as in K_p -partitioning (Definition 2.2). However, notice that in a (k, ν) -balanced partitioning problem given as input is the number of partitions, whereas in a K_p -partitioning or a bounded-size partitioning problem (Definition 2.3) given as input is the size (or an upper bound on the size) of each partition.

Andreev and Harald [4] showed, by reducing from the 3-Partition problem, that the general $(k, 1)$ -balanced partitioning (where k is not limited to being constant) is **NP**-hard to approximate within any finite approximation factor. However, a careful inspection of their reduction reveals that it requires n/k (the size of each partition) not be bounded by a constant as well. Therefore, the hardness result of Andreev and Harald cannot settle our variation of the problem (Definition 2.3).

On the positive side, Saran and Vazirani [32] gave the first nontrivial approximation algorithm for the $(k, 1)$ -balanced graph partitioning problem, for a constant k . Their algorithm is based on the Gomory-Hu tree that encodes in itself the minimum s - t cut for all (s, t) pairs of vertices in the original graph. The Gomory-Hu Tree is formally defined as follows.

Definition 3.2 (Gomory-Hu Tree [16]). *For a graph $G(V, E)$, $T(V, E')$ is the Gomory-Hu tree of G if and only if for all pairs $(s, t) \in V$, the weight of the minimum edge along the s - t path in T is equal to the minimum s - t cut in G .*

Gomory, Hu, and Chung [16] showed that such a tree exists for every graph and can be found in polynomial time. The cuts associated with edges of the tree are called Gomory-Hu cuts.

The approximation algorithm for the $(k, 1)$ -balanced partitioning is described as follows. First it finds a set of $n - 1$ Gomory-Hu cuts in the graph, and sorts them by increasing weight, obtaining $g_1, g_2, \dots, g_{n-1}$. Then it finds the minimum i such that the connected components in $G - (g_1 \cup \dots \cup g_i)$ can be partitioned into k sets of size n/k each. For a constant k , this can be achieved using a polynomial time dynamic programming approach. Saran and Vazirani showed that this algorithm delivers a tight approximation factor of $(\frac{k-1}{k})n < n$.

To adapt this algorithm to our bounded-size partitioning problem, we notice that in bounded-size partitioning, we merely need to partition the vertices into disjoint sets with their size bounded by p . Thus we only need to find the smallest i such that every connected component in $G - (g_1 \cup \dots \cup g_i)$ has at most p vertices. It follows via an analogous analysis that this algorithm gives an approximation factor of n . Even though the approximation factor is not satisfactory, it does imply that the problem admits a finite approximation factor algorithm and in turn, brings our problem out of the scope of the above-mentioned inapproximability result of Andreev and Harald. Thus we have the following corollary.

Corollary 3.1. *For the one-block cache, and for every packing factor p , the data packing problem can be approximated within a factor of n , where n is the number of distinct objects in the reference sequence.*

Feige and Krauthgamer [12] devised an algorithm for the $(k, 1)$ -balanced partitioning that gives an approximation ratio of $O(\log^{1.5} n)$. However, their algorithm requires a running time that is exponential in k , and therefore, cannot be applied to our problem. In fact, even for the $(2, 1)$ -balanced graph partition problem, which is also known as **MINIMUM BISECTION** problem, it is not known whether a constant factor approximation exists, although several results might indicate that it cannot be approximated arbitrarily well in polynomial time [6, 11].

4. Approximation for m -Block Cache

It is reasonable to think that as the number of cache blocks increases, the data packing problem becomes more difficult. In this section, we prove several hardness results regarding the m -block cache.

Our first and most general result is that for every number of blocks in the cache starting at 5, the problem is NP-hard to approximate even within an approximation factor of $O(N^{1-\epsilon})$ for any $\epsilon > 0$, where N is the length of the reference sequence. In our proof, we will use a reduction from the well-known **3SAT** problem. Then, we will use the same approach to prove that, based on a hardness result for the **2SAT-MIN-DEL** problem, for any number of blocks in the cache starting at 3, the existence of a constant factor approximation algorithm for the data packing problem disproves the so-called “Unique Games Conjecture.”

Our hardness results are general and hold for all *sensible* replacement policies. The notion of “sensible replacement policies” was first defined by Petrank and Rawitz [27] for t -way set associative caches, and it includes all replacement policies that do not perform extremely bad. Here, we adapt their definition to our data packing problem.

Definition 4.1 (*e*-sensible replacement policies for fully-associative cache (adapted from [27])). *Let m be the number of blocks in the cache. For a fixed constant e , a replacement policy is called *e*-sensible if the following holds. For every two reference sequences σ and σ' , and every integer ℓ , if the referenced objects in σ are packed in a total of $q \leq m$ blocks, then the sequence $\sigma' \odot \sigma^\ell$ (σ' followed by ℓ iterations of σ) incurs at most $e + q$ cache misses over the accesses in σ^ℓ .*

As Petrank and Rawitz observed, replacement policies such as LRU, FIFO, and pseudo-LRU are all 0-sensible replacement policies. Replacement algorithms employed by software-managed caches often have higher performances and therefore, are 0-sensible as well.

We now state and prove our most general result regarding the inapproximability of data packing for any number of blocks in the cache starting at 5.

Theorem 4.1. *For every number of blocks in the cache $m \geq 5$, every packing factor $p \geq 2$, and under any sensible replacement policy, no polynomial algorithm can approximate the data packing problem within a factor of $O(N^{1-\epsilon})$ for any constant $\epsilon > 0$, unless $P = NP$ (N is the length of the reference sequence).*

Proof. Consider any *e*-sensible replacement policy, for an arbitrary constant e . We reduce the following **3SAT** problem to a data packing problem under this replacement policy, and with m blocks in the cache and packing factor p .

Definition 4.2 (3SAT). *Given a set $\mathcal{V}$ of variables, and a collection $\mathcal{C}$ of clauses over $\mathcal{V}$ such that each clause $c \in \mathcal{C}$ has exactly three*

literals, is there a truth assignment for $\mathcal{V}$ that satisfies every clause in $\mathcal{C}$?

Let $\delta = \epsilon/4$. We will ensure in our reduction that if we reduce from a positive instance of **3SAT**, then the optimal number of misses is $o(N^{2\delta})$, and if we reduce from a negative instance, then the optimal number of misses is $\omega(N^{1-2\delta})$. Therefore, if a polynomial approximation algorithm exists for the data packing problem, and that algorithm is able to find a solution that is always within a factor of $O(N^{1-4\delta})$ of the optimal solution, then we can solve the **3SAT** problem in polynomial time, by testing if the value of the approximate solution is $\omega(N^{1-2\delta})$.

Let us first introduce some notations that we will use during the construction. We will generate a reference sequence by concatenating smaller sequences which we introduce during our construction. We will use $A \odot B$ to denote the concatenation of sequences A and B , and use A^ℓ to denote ℓ iterations of the sequence A . For a data packing scheme P , we use the notation $(a, b) \in P$ to denote that P packs the sets or objects a and b in the same memory block. Finally, for a set of data objects D , we let $Seq(D)$ be any fixed linear arrangement of objects in D . Notation-wise, $Seq(D) = \bigodot_{d \in D} d$. We denote by $Miss_{opt}$ the number of misses that our reference sequence incurs under the optimal data packing scheme.

Given an instance of the **3SAT** problem, we construct an instance of the data packing problem as follows.

Let $f = (|\mathcal{C}| + |\mathcal{V}|)^\ell$. Later on, we will determine ℓ in a way to ensure that under any approximate data packing scheme, our reference sequence incurs less than f misses if and only if we map from a positive **3SAT** problem.

We construct the reference sequence as follows. We will have two auxiliary sets of data objects in our data packing problem instance:

- $R = \{r_1, r_2, \dots, r_{p(m-2)}\}$
- $S = \{s_1, s_2, \dots, s_{p(m-5)}\}$

We ensure that if the minimal miss count $Miss_{opt} < f$, then any optimal packing scheme will pack the objects in each of these two sets together.

To ensure this for objects in R , we add the sequences

$$(r_1^* \odot r_2^* \odot \dots \odot r_{2p}^* \odot Seq(R))^f$$

and

$$(r_{2p+1}^* \odot r_{2p+2}^* \odot \dots \odot r_{4p}^* \odot Seq(R))^f,$$

where $r_1^*, r_2^*, \dots, r_{4p}^*$ are auxiliary object which will not be accessed elsewhere in the sequence.

If any object in R is packed with an object not in R , then at least in one of the two sequences above, we will have at least one miss in every iteration, leading to at least f misses, which contradicts our assumption. However, if all the objects in R are packed together and we have $(r_1^*, r_2^*, \dots, r_{2p}^*)$ and $(r_{2p+1}^*, r_{2p+2}^*, \dots, r_{4p}^*)$ as packed groups, then at most $e+m$ misses will occur in each of the above sequences.

We will use the sequence $Seq(R)$ in the sequences corresponding to variables of the **3SAT** problem.

Analogously, we ensure that if $Miss_{opt} < f$, then any optimal packing scheme will pack the objects in S together. To do so, we add the sequences

$$(s_1^* \odot s_2^* \odot \dots \odot s_{5p}^* \odot Seq(S))^f$$

and

$$(s_{5p+1}^* \odot s_{5p+2}^* \odot \dots \odot s_{10p}^* \odot Seq(S))^f,$$

where $s_1^*, s_2^*, \dots, s_{10p}^*$ are auxiliary objects which will not be accessed elsewhere in the sequence and $f = (|\mathcal{V}| + |\mathcal{C}|)^\ell$ as before.

We will use the sequence $Seq(S)$ in the sequences corresponding to clauses of the **3SAT** problem.

For every variable u in the **3SAT** instance, we introduce $3p$ objects in our data packing instance:

- $A^u = \{a_1^u, a_2^u, \dots, a_p^u\}$,
- $B^u = \{b_1^u, b_2^u, \dots, b_p^u\}$, and
- $C^u = \{c_1^u, c_2^u, \dots, c_p^u\}$.

Let $\overline{A}^u = A^u - \{a_1^u\}$, $\overline{B}^u = B^u - \{b_1^u\}$, and $\overline{C}^u = C^u - \{c_1^u\}$.

For brevity in presentation, and since we are currently dealing only with objects corresponding to a single variable, we drop the superscript u for these objects, e.g., we refer to a_i^u and A^u , respectively by a_i and A . Later, when we construct the sequences corresponding to the clauses, we will add these superscripts again.

We will ensure that, if $\text{Miss}_{\text{opt}} < f$, any optimal data packing scheme will group these objects in one of the two following ways (we call the first packing packing I and the second packing packing II).

1. $(a_1, \overline{A}), (b_1, \overline{B}), (c_1, \overline{C})$
2. $(a_1, \overline{B}), (b_1, \overline{C}), (c_1, \overline{A})$

To do so, we append the following six sequences to our reference sequence.

1. $(a_1 \odot b_1 \odot \text{Seq}(\overline{B}) \odot \text{Seq}(R))^f$
2. $(b_1 \odot c_1 \odot \text{Seq}(\overline{C}) \odot \text{Seq}(R))^f$
3. $(c_1 \odot a_1 \odot \text{Seq}(\overline{A}) \odot \text{Seq}(R))^f$
4. $(\text{Seq}(\overline{A}) \odot \text{Seq}(\overline{B}) \odot a_1 \odot \text{Seq}(R))^f$
5. $(\text{Seq}(\overline{B}) \odot \text{Seq}(\overline{C}) \odot b_1 \odot \text{Seq}(R))^f$
6. $(\text{Seq}(\overline{C}) \odot \text{Seq}(\overline{A}) \odot c_1 \odot \text{Seq}(R))^f$

In the following discussion, we assume that our data packing instance admits a packing scheme which leads to less than f misses. That is $\text{Miss}_{\text{opt}} < f$. Let us call the objects in $A \cup B \cup C$ the primary objects. Each of the above sequences makes references to at least $p + 1$ primary objects, which will be packed in a total of two blocks at least. However, if these referenced objects are packed in more than two blocks, the corresponding sequence will incur at least one miss in every iteration, giving a total of at least f misses, which contradicts our assumption. Thus under any approximate packing solution, the primary objects referenced by each of these sequences must be placed in a total of two blocks.

The above observation implies that under any optimal packing solution, the data objects in each of $\overline{A}$, $\overline{B}$, and $\overline{C}$ cannot be divided among more than two blocks. We further show that each of $\overline{A}$, $\overline{B}$, and $\overline{C}$ is indivisible. That is, under any optimal packing scheme, their objects must reside in a single block. Otherwise, let us assume without loss of generality (as the other cases are symmetric) that the objects in $\overline{A}$ are divided between two blocks. This leaves $p + 1$ vacant positions in those blocks. Sequences 4 and 6, together enforce that the objects in $\{a_1\} \cup \overline{B} \cup C$ must reside in those $p + 1$ vacant positions, which is impossible since $|\{a_1\} \cup \overline{B} \cup C| = 2p > p + 1$. By symmetry, this argument holds for $\overline{B}$ and $\overline{C}$ as well.

We further argue that any optimal packing scheme P cannot pack any two objects from $\{a_1, b_1, c_1\}$ in the same block. Otherwise, let us assume without loss of generality (as the other cases are symmetric) that $(a_1, b_1) \in P$. Neither $\overline{A}$ nor $\overline{B}$ can be packed in the same block as a_1 and b_1 . So sequence 4 enforces that $(\overline{A}, \overline{B}) \in P$. On the other hand, from sequence 3 we know that the objects in $A \cup \{c_1\}$ must be packed in a total of two blocks. However, P packs a_1 with b_2 , and $\overline{A}$ with $\overline{B}$. This implies that $\|A \cup B \cup \{c_1\}\| \leq 2p$ which is a contradiction.

Now let us assume that for an approximate packing scheme P , we have $(a_1, \overline{A}) \notin P$. Due to sequence 3, and since $(a_1, c_1) \notin P$,

we must have $(c_1, \overline{A}) \in P$. Consequently, from sequence 4, we must have $(a_1, \overline{B}) \in P$. This results in packing I, which leads to $m + e$ misses for each of the six sequences that we described above.

On the other hand, if $(a_1, \overline{A}) \in P$, then sequences 1 and 2 respectively enforce $(b_1, \overline{B}) \in P$ and $(c_1, \overline{C}) \in P$, resulting in packing II. This packing also leads to $m + e$ misses for each of the six sequences.

We will associate packing I with the variable u being assigned to true and packing II with this variable being assigned to false.

For every variable u , let h^u refer to the object a_1^u and let h^{-u} refer to the object c_1^u . Additionally, let A^{-u} be an alias for the set A^u . Now, for every clause $c = x \vee y \vee z$ in the **3SAT** instance, where x, y , and z are arbitrary literals, we add the following sequence to our reference sequence.

$$(h^x, \text{Seq}(\overline{A}^x), h^y, \text{Seq}(\overline{A}^y), h^z, \text{Seq}(\overline{A}^z), \text{Seq}(S))^f$$

Under a packing scheme, if the first $3p$ objects in this sequence exceed 5 blocks of memory, then every iteration of the sequence incurs at least one miss. However, if these objects occupy no more than 5 blocks, then the sequence would incur no more than $m + e$ misses. Under a packing P , these $3p$ objects are packed in no more than 5 blocks if and only if $\{(h^x, A^x), (h^y, A^y), (h^z, A^z)\} \cap P \neq \emptyset$.

This completes the description of the sequences that compose the reference sequence in our data packing problem instance.

We now determine ℓ such that under any approximate packing scheme, less than f misses occur if and only if we map from a positive **3SAT** problem instance.

If $\text{Miss}_{\text{opt}} < f$, then consider an arbitrary optimal packing scheme P . For every one of the above sequences, P must pack their referenced objects in a total of at most m blocks. Otherwise, that sequence incurs at least one miss in every iteration, which results in f misses at least. Thus under P , each sequence incurs a total of at most $m + e$ misses. Since we introduced a total of $6|C| + |\mathcal{V}| + 4$ sequences, the reference sequence incurs a total of at most $(m + e)(6|C| + |\mathcal{V}| + 4)$ misses. This implies that $\text{Miss}_{\text{opt}} \leq (m + e)(6|C| + |\mathcal{V}| + 4)$.

This implies that for the data packing instance that we construct, one of the two following cases can happen.

1. $\text{Miss}_{\text{opt}} \leq \text{Miss}_{\text{small}} = (m + e)(6|C| + |\mathcal{V}| + 4)$
2. $\text{Miss}_{\text{opt}} \geq \text{Miss}_{\text{big}} = f = (|C| + |\mathcal{V}|)^\ell$

We claim that if we map from a positive instance of the **3SAT** problem, our mapped-to data packing instance falls into the first case. Consider a satisfying assignment for the **3SAT** instance. For every variable u in the **3SAT** instance, if u is true under that satisfying assignment, we pack the objects in $A^u \cup B^u \cup C^u$ according to packing I. Otherwise, we pack them according to packing II. Additionally, we pack the objects in R together, and the objects in S together as well, so that their corresponding four sequences incur no more than $m + e$ misses. It is easy to verify that each of the introduced sequences will incur at most $m + e$ misses under this packing and therefore, $\text{Miss}_{\text{opt}} \leq \text{Miss}_{\text{small}}$.

On the other hand, we show that if we map from a negative instance of the **3SAT** problem, we map to a data packing instance that falls into the second case. That is because, by our construction, a packing scheme either corresponds to a truth assignment for the mapped-from **3SAT** instance or leads to at least f misses. However, if the **3SAT** instance is not satisfiable, then even under a packing scheme that corresponds to a truth assignment, there is at least one sequence (corresponding to a clause) that incurs at least f misses. Thus we have $\text{Miss}_{\text{opt}} \geq \text{Miss}_{\text{big}}$.

The total number of accesses in the reference sequence is equal to

$$N = (4pm + 6(p(m-1) + 1)|\mathcal{C}| + p(m-2)|\mathcal{V}|)(|\mathcal{C}| + |\mathcal{V}|)^\ell = \Theta((|\mathcal{C}| + |\mathcal{V}|)^{\ell+1}).$$

Thus if we choose $\ell = \frac{1-\delta}{\delta}$, we have

$$Miss_{big} = (|\mathcal{C}| + |\mathcal{V}|)^\ell = N^{1-\delta} = \omega(N^{1-2\delta})$$

and

$$Miss_{small} = O(|\mathcal{C}| + |\mathcal{V}|) = O(N^\delta) = o(N^{2\delta}).$$

Now let us assume that for the data packing problem, we have an approximation algorithm that gives a data packing scheme which is always within an $O(N^{1-4\delta})$ factor of the optimal. Using this algorithm, we can solve the **3SAT** problem as follows. For every **3SAT** instance I , we run the algorithm on the data packing instance that we construct as above. Let $Miss_{approx}$ be the value of the approximate solution. That is the number of misses incurred by our reference sequence after applying the approximate packing scheme. One of the two cases can happen.

- I is satisfiable: In this case, we have

$$\begin{aligned} Miss_{approx} &= O(N^{1-4\delta} \times Miss_{opt}) \\ &= O(N^{1-4\delta} \times Miss_{small}) \\ &= o(N^{1-4\delta} \cdot N^{2\delta}) = o(N^{1-2\delta}). \end{aligned}$$

- I is not satisfiable: In this case, we have

$$Miss_{approx} \geq Miss_{opt} \geq Miss_{big} = \omega(N^{1-2\delta}).$$

Thus we can distinguish between a positive and a negative **3SAT** instance, and this implies $\mathbf{P} = \mathbf{NP}$. This completes our proof. $\square$

The reference sequence constructed in our reduction can be viewed as an actual data access trace in a program. This program consists of $O(N^\epsilon)$ phases, each comprising of $\Omega(N^{1-\epsilon})$ iterations over a constant number of data objects. This implies that our hardness of approximation result is valid, even when restricted to reference sequences with $O(N^\epsilon)$ phases. This suggests that to be able to efficiently solve the data packing problem, we must turn our focus to the case where the program's phase behavior changes at a rate that is lower than any polynomial in the size of the trace. We leave this as future work.

Looking from another perspective, it is interesting to find out whether, if the number of blocks in the cache drops below 5, efficient approximations exist for the data packing problem. Next, we will prove a weaker hardness result for when the number of cache blocks is either 3 or 4. This result is based on the so-called **Unique Games Conjecture** [21].

Theorem 4.2. *For every number of cache blocks $m \geq 3$ and every packing factor $p \geq 2$, and under any sensible replacement policy, no polynomial algorithm can approximate the data packing problem within any constant approximation factor, unless the Unique Games Conjecture is false.*

Proof. Khot [21] showed that if the so-called “Unique Games Conjecture” holds then the following problem is **NP**-hard to approximate within any constant factor.

Definition 4.3 (2SAT Minimum Deletion (2SAT-MIN-DEL)). *Given a set $\mathcal{V}$ of variables and a collection $\mathcal{C}$ of clauses over $\mathcal{V}$ such that each clause in $\mathcal{C}$ has exactly two literals, find the smallest subset of clauses $\mathcal{C}'$ such that $\mathcal{C} - \mathcal{C}'$ is satisfiable.*

Consider any e -sensible replacement policy. We slightly modify our reduction used in the proof of Theorem 4.1 to reduce from **2SAT-MIN-DEL** instead. We show that for any number of cache blocks m the existence of a k -factor approximation algorithm for the data packing problem implies a $2k(m+e+1)$ -factor approximation algorithm for the **2SAT-MIN-DEL** problem.

The construction will almost be the same as the one in the proof of Theorem 4.1. We will have the same auxiliary sets of objects R and S except that S will include $p(m-3)$ objects. Using the same approach as before, this time we set f to ensure that any packing solution with a constant factor approximation ratio will pack the objects in these sets together. To this end, for each of S and R we add the analogous two bookkeeping sequences, except that for these sequences, we set

$$f = k(|\mathcal{C}| + 1)(m + e + 1)(6|\mathcal{C}| + |\mathcal{V}| + 4) + 1.$$

Let us throughout this proof use the shorthand notation $\theta = 6|\mathcal{C}| + |\mathcal{V}| + 4$.

We construct the sequences corresponding to variables as before. We use $f = k(|\mathcal{C}| + 1)(m + e + 1)\theta + 1$ for each of the 6 sequences, corresponding to each variable.

The sequences corresponding to clauses are constructed in the same way, except that since every clause in the **2SAT-MIN-DEL** problem has two literals, each of these sequences will include $2p$ objects besides the objects in S . For these sequences, we set $f = \theta$.

Under a packing scheme that corresponds to a truth assignment for the **2SAT-MIN-DEL** instance, every reference subsequence, except the ones introduced for clauses, incurs at most $m + e$ misses (according to the definition of e -sensible replacement policies). Additionally, for every non-satisfied clause, its corresponding sequence incurs at most $(m + 1)\theta$ and at least θ misses. Thus if the truth assignment satisfies all but g clauses, the reference sequence incurs an overall of at least $g\theta$ and at most

$$g(m + 1)\theta + (m + e)(6|\mathcal{C}| + 4) \leq (g + 1)(m + e + 1)\theta$$

misses.

On the other hand, under a packing scheme that does not correspond to a truth assignment for the variables in the instance of **2SAT-MIN-DEL**, the reference sequence incurs at least $k(|\mathcal{C}| + 1)(m + e + 1)\theta + 1$ misses.

Now suppose there exists a k -approximation algorithm for the data packing problem. We obtain a $2k(m + e + 1)$ -approximation algorithm for the **2SAT-MIN-DEL** problem as follows. Given a **2SAT-MIN-DEL** instance, let us denote by g_{opt} the number of non-satisfied clauses under the optimal truth assignment. Since **2SAT** $\in \mathbf{P}$ we can decide in polynomial time if $g_{opt} = 0$. So let us without loss of generality assume that $g_{opt} \geq 1$. We use the reduction that we described above to construct a data packing problem instance. For this instance, let us denote by $Miss_{approx}$ the number of misses under the approximate data packing scheme, and by $Miss_{opt}$ the number of misses under the optimal data packing scheme. A data packing scheme that corresponds to an optimal truth assignment leads to at most $(g_{opt} + 1)(m + e + 1)\theta$ misses. Thus we have

$$Miss_{opt} \leq (g_{opt} + 1)(m + e + 1)\theta \leq (|\mathcal{C}| + 1)(m + e + 1)\theta$$

and therefore,

$$Miss_{approx} \leq k(|\mathcal{C}| + 1)(m + e + 1)\theta.$$

This implies that the approximate and the optimal data packing schemes must correspond to valid truth assignments for the **2SAT-MIN-DEL** instance.

Let us denote by g_{approx} the number of non-satisfied clauses by the truth assignment corresponding to the approximate data packing solution. We have

$$g_{approx}\theta \leq Miss_{approx}.$$

Finally,

$$\begin{aligned}
g_{\text{approx}} &\leq \frac{\text{Miss}_{\text{approx}}}{\theta} \leq \frac{k \text{Miss}_{\text{opt}}}{\theta} \\
&\leq \frac{k(g_{\text{opt}} + 1)(m + e + 1)\theta}{\theta} \\
&= k(g_{\text{opt}} + 1)(m + e + 1) \leq 2k(m + e + 1)g_{\text{opt}},
\end{aligned}$$

where the last inequality follows our assumption that $g_{\text{opt}} \geq 1$. This means that we can use the approximate packing scheme to find a truth assignment for the **2SAT-MIN-DEL** instance with a value that is within a constant factor of the optimal. This breaks the Unique Games Conjecture.

This completes the proof. $\square$

5. Approximating the Cache Hit Maximization Problem

For the data packing problem, we can take as the objective either maximizing the cache hits or minimizing the cache misses. Even though the two choices result in the same optimization problem, they can have different implications in terms of approximation. Our hardness result for the miss minimization problem (Theorem 4.1) establishes a computational barrier for this problem, but does not shed light on the approximability of the hit maximization problem. For this problem, a constant factor approximation for all cache sizes would be a breakthrough result, even from a practical point of view. Here, we establish a fixed approximation factor for every fixed number of blocks in the cache. Our analysis and discussions expose the challenges ahead in designing a constant factor approximation algorithm for all cache sizes.

We showed in section 2 that for the one-block cache, the data packing problem can be formulated as a bounded-size graph partitioning problem (Definition 2.3). Analogously, for this cache, the cache hit maximization version of the problem can be formulated as the dual problem of bounded-size graph partitioning, which is known as the **GRAPH PACKING** problem. This problem asks for partitioning the graph into bounded-size sets while maximizing the total weight of edges whose endpoints are in the same partition. For a packing factor of 2, the problem is equivalent to maximum weighted matching and can be solved *exactly* in polynomial time. However, for larger packing factors, the problem becomes **NP-hard**, and therefore, the approximation problem becomes interesting.

Obviously, every approximation algorithm for the graph packing problem is an approximation for the cache hit maximization problem in a one-block cache. In fact, we go a bit further and describe an approximation algorithm to solve the hit maximization problem in the m -block LRU cache, and for a packing factor of p , within an approximation factor of mp . Instead of the graph packing problem, our algorithm will use an approximation algorithm for the more general problem of **Maximum Weighted p -Set Packing** (also known as hypergraph packing), which is defined as follows.

Definition 5.1 (Maximum Weighted p -Set Packing [5] (Hypergraph Packing)). *Given a collection $\mathcal{F}$ of sets, each containing at most p elements along with their integer weights, find a collection of disjoint sets in $\mathcal{F}$ which has the maximum total weight.*

A straightforward greedy algorithm for this problem iteratively picks the set with the maximum weight until no other set can be picked. It is easy to show that this algorithm gives an approximation ratio of p . Improving over this approach, Arkin and Hassin [5] gave a local search algorithm that can achieve an approximation factor of $k - 1 + \frac{1}{t}$ for every constant t . Chandra and Halldórsson [9] improved this to get an approximation factor of $2(k + 1)/3$.

We first introduce some notations that will help us describe and analyze the algorithm. For a set S , we use $\|S\|$ to denote the

cardinality of S . For every reference x in the reference sequence, we denote by d_x the data object referenced by x . For two references x and y in the reference sequence, we denote by $\text{volume}(x, y)$ the set of data objects referenced between, and not including x and y . We alternatively call $\text{volume}(x, y)$ the volume distance between x and y . We denote by $x \prec y$ the fact that x comes before y in the reference sequence, and by $x \preceq y$ the fact that either $x = y$ or $x \prec y$.

Consider an arbitrary packing P . For two references x and y , such that $x \prec y$, we denote by $\text{volume}_P(x, y)$ the set of all **memory blocks** referenced between, and not including x and y .

Finally, for two reference $x, y \in T$, such that $x \prec y$, and for a data packing scheme P , (x, y) is a **block reuse interval** or simply, a reuse interval under P if the following three conditions hold.

1. Under the packing P , the objects d_x and d_y are packed in the same memory block. That is

$$(d_x, d_y) \in P.$$

2. For every data object c accessed between x and y , P does not pack c in the same block as d_x and d_y . That is, if B is the block containing d_x and d_y , then

$$B \notin \text{volume}_P(x, y).$$

3. Under the packing P , the number of distinct **blocks** referenced between x and y is less than m (number of blocks in the cache), i.e.,

$$\|\text{volume}_P(x, y)\| < m.$$

The above conditions ensure that a reference y is the end of a reuse interval under the packing P if and only if applying the packing P makes the reference y hit in the cache. Furthermore, every reference can be the end of no more than one reuse interval. Thus under any packing, every cache hit corresponds to one and only one reuse interval and vice versa. Therefore, maximizing the cache hits is equivalent to maximizing the number of reuse intervals.

For two references x and y , respectively to objects a and b , if $\text{volume}(x, y) \geq m$, then packing a and b together is not sufficient to guarantee that (x, y) is a block reuse interval. In such a case, it is necessary to pack some pairs in $\text{volume}(x, y)$ to ensure that $\text{volume}_P(x, y) < m$. Because of this interdependence between block reuse intervals, it is ambitious to design an algorithm that would directly aim at maximizing the number of block reuse intervals. Our algorithm will instead target at maximizing a more specific type of reuse intervals, which we call **confident reuse intervals**.

We define confident reuse intervals as follows. Under a packing scheme P , and for two references $x, y \in T$, such that $x \prec y$, (x, y) is a **confident** (block) reuse interval under P if

1. (x, y) is a reuse interval under P , and
2. $\|\text{volume}(x, y)\| < m$.

In other words, a reuse interval is confident if it includes references to less than m objects. By “confident” we mean that the cache hit at y is independent of how the objects in $\text{volume}(x, y)$ are packed and relies only on the packing of the objects d_x and d_y in the same block.

Our algorithm formulates the **confident reuse interval maximization problem** as a weighted p -set packing problem. Given the sequence T , let $\mathcal{U}$ be the set of data elements referenced in T . We construct an instance of the weighted p -set packing problem as follows. The collection of sets in our set packing problem includes every subset of $\mathcal{U}$ with at most p elements (all p -subsets of $\mathcal{U}$). For a p -subset S , we set the weight of S in our set packing problem equal to

$$\begin{aligned}
w(S) &= \|\{(x, y); \quad x, y \in T, x \prec y, \\
&\quad d_x, d_y \in S, \\
&\quad volume(x, y) \cap S = \phi, \\
&\quad \|volume(x, y)\| < m\}\|.
\end{aligned}$$

In other words, consider all the intervals that begin and end with references to elements in S . The weight of the set S is equal to the number of such intervals that (i) include no references to any other element in S , and (ii) include references to at most $m - 1$ distinct objects.

Example 5.1. Consider the following reference sequence.

$$a \quad u \quad v \quad \underbrace{b \quad c}_{v} \quad b$$

For a 2-block cache and a packing factor of 3, there are six intervals beginning and ending with references to elements in the set $\{a, b, c\}$. Only the intervals that are shown by braces contribute to the weight of this set and therefore, $w(\{a, b, c\}) = 2$.

The collection of weighted sets in our set packing problem generalizes the proximity graph (as defined by Thabit [36] for the one-block cache) to caches of arbitrary size. So we alternatively call this collection the **proximity hypergraph**.

It is easy to verify that an optimal packing for the constructed proximity hypergraph gives an optimal packing scheme for the confident reuse interval problem and vice versa. So we can use an α -approximation algorithm for the p -set packing problem to solve the confident reuse interval problem within an approximation factor of α .

We now state and prove that for the general cache hit maximization problem, the above algorithm achieves an approximation factor of αm .

Theorem 5.1. For an m -block cache, and a packing factor of p , and given an α -approximation algorithm for the weighted p -set packing problem, the above algorithm finds a data packing scheme under which the number of cache hits (block reuse intervals) is within a factor of αm of the optimal.

Proof. We show that an α -approximate packing solution for the confident reuse interval problem is in fact an αm -approximate solution for the original cache hit maximization problem. We argue that under any packing, the number of reuse intervals is at most a factor of m away from the number of confident reuse intervals.

Under a packing P , we say that a reuse interval (x, y) **encloses** a confident reuse interval (z, t) if $x \preceq z \prec t \preceq y$. We argue that under any packing, every reuse interval encloses at least one confident reuse interval and every confident reuse interval is enclosed by at most m different reuse intervals.

For a packing P , let (x, y) be a (not necessarily confident) reuse interval. We first show that there exists at least one pair of references z and t , such that $x \preceq z \prec t \preceq y$ and (z, t) is a confident reuse interval under P .

If $\|volume(x, y)\| < m$, then (x, y) itself is a confident reuse interval. So we can set $z = x$ and $t = y$ and we are done. Otherwise if $\|volume(x, y)\| \geq m$, let h be the earliest reference after x such that $\|volume(x, h)\| = m$. Clearly $x \prec h \preceq y$. If P packs all the objects in $volume(x, h)$ in separate blocks, then we have $\|volume_P(x, h)\| \geq m$, which contradicts the assumption that (x, y) is a reuse interval. So there are at least two objects $c, d \in volume(x, h)$ which are packed together by P in the same memory block E . Let z and t be two references to the block E such that $x \prec z \prec t \preceq h$ and $E \notin volume_P(z, t)$. (z, t) is clearly

a reuse interval. Furthermore, since $volume(z, t)$ does not include the object referenced by z , we have

$$volume(z, t) \subset volume(x, h)$$

and therefore, $\|volume(z, t)\| < m$. Thus (z, t) is a confident reuse interval.

Next we show that for every x and y , such that $x \prec y$, (x, y) is enclosed by no more than m reuse intervals. Let (u, v) and (u', v') be two reuse intervals enclosing (x, y) . We argue that (u, v) and (u', v') cannot both refer to the same block. Otherwise, since the two intervals overlap, at least one of the following cases would happen.

1. $volume(u, v) \cap \{d_{u'}, d_{v'}\} \neq \phi$
2. $volume(u', v') \cap \{d_u, d_v\} \neq \phi$

This contradicts the assumption that both (u, v) and (u', v') are reuse intervals. Thus every reuse interval enclosing (x, y) must refer to a distinct block.

Consider one reuse interval (u, v) enclosing (x, y) . Let B be the block referenced by u and v . During the reuse interval defined by (u, v) the sequence makes no other references to B . Thus the cache must hold B during the entire (u, v) interval. On the other hand, the intersection of the intervals enclosing (x, y) is nonempty (at least covers the interval (x, y)). This implies that at some point in time, the cache must hold all the blocks corresponding to the reuse intervals enclosing (x, y) . Since every enclosing interval must refer to a distinct block, the total number of reuse intervals enclosing (x, y) is at most m .

Our arguments above indicate that under any packing, every reuse interval encloses at least one confident reuse interval and every confident reuse interval is at most enclosed by m reuse intervals. This means that under any packing, the number of reuse intervals is at most m times the number of confident reuse intervals. Therefore, an approximation factor of α for the confident reuse problem gives an approximation factor of αm for the original cache hit maximization problem.

This completes the proof that given an α -approximation algorithm for the weighted p -set packing problem, our algorithm achieves an approximation factor of αm for maximizing the cache hits, for an m -block cache and packing factor p . $\square$

Notably, for the special case when packing factor is 2, since the weighted matching problem can exactly be solved in polynomial time, our algorithm can achieve an approximation factor of m . For general packing factor $p \geq 3$, we can use the $2(p+1)/3$ -approximation algorithm for the set packing problem and achieve an approximation factor of $2(p+1)m/3$ for cache hit maximization.

One might naturally wonder whether the approximation factor that we established above is tight. Our approximation factor consists of two multiplicative terms: the number of blocks in the cache and the approximation factor for set packing. Hazan et al. [18] have shown that even the unweighted p -set packing problem is **NP**-hard to approximate within a factor of $p/\ln p$. So the approximation factor for set packing is almost tight. Next we show that our ratio is asymptotically tight with respect to the number of blocks in the cache, as well.

Theorem 5.2. For $p = 2$ and for an m -block cache, our algorithm cannot deliver a performance guarantee better than $2/m$.

Proof. We construct an infinite number of data packing instances with $p = 2$, such that the algorithm cannot deliver a performance guarantee better than $2/m$, on any of these instances.

For every $i \in \{1, 2, \dots, m-1\}$ we have two sets each containing t objects.

- $\mathcal{A}_i = \{a_i^1, a_i^2, \dots, a_i^t\}$
- $\mathcal{B}_i = \{b_i^1, b_i^2, \dots, b_i^t\}$

We call the set $\mathcal{C}_i = \mathcal{A}_i \cup \mathcal{B}_i$ the i th bucket.

We first build a graph H that will help us construct the reference sequence. For every tuple $u = (u_1, u_2, \dots, u_{m-1}) \in \{1, 2, \dots, t\}^{m-1}$, we add a node u in the graph. For every two tuples $u = (u_1, u_2, \dots, u_{m-1})$ and $v = (v_1, v_2, \dots, v_{m-1})$, we add to H , directed edges from u to v and from v to u if u and v differ exactly in one component. That is, if there exists $j \in \{1, 2, \dots, m-1\}$ such that for every i we have

$$u_i = v_i \Leftrightarrow i \neq j.$$

The out-degree and in-degree of every vertex in H is equal to $(m-1)(t-1)$. So H has an Eulerian tour. Let $T = \nu_1 \nu_2 \nu_3 \dots \nu_1$ be a traversal of the vertices of H according to an Eulerian tour in G . The length of this tour is equal to the number of edges in H , i.e., $(m-1)(t-1)t^{m-1}$. For every vertex $u = (u_1, u_2, \dots, u_{m-1})$ appearing in T , we replace u in T with the following sequence. (For ease of presentation, we assume that our reference sequence is circular. This assumption will not affect the performance guarantee.)

$$a_1^{u_1} \odot a_2^{u_2} \odot \dots \odot a_{m-1}^{u_{m-1}} \odot x \odot y \odot b_1^{u_1} \odot b_2^{u_2} \odot \dots \odot b_{m-1}^{u_{m-1}} \odot x \odot y$$

Since $p = 2$, our algorithm will effectively construct a weighted graph and outputs its maximum matching. Let us determine the weight of the edges in this graph. Let $w(e, f)$ be the weight of the edge (e, f) in this graph.

First, every two consecutive accesses to x and y occur within a volume distance of at most $m-1$. Thus we have

$$w(x, y) = 4(m-1)(t-1)t^{m-1}.$$

Second, for every object $q \notin \{x, y\}$, whenever q is accessed in the trace it is enclosed between two accesses to x and two accesses to y . Moreover, the volume distance between the access to q and the four accesses to x and y is less than m . Thus for every object q we have

$$w(q, x) = w(q, y) = 2(m-1)(t-1)t^{m-2}.$$

Thus we can take t large enough to ensure that x and y will be matched together in any maximum matching.

Third, it is easy to verify that for every bucket $\mathcal{C}_i$, every two objects from that bucket are always accessed within a volume distance of at least m . Thus for every $i \in \{1, 2, \dots, m-1\}$ and for every two objects $c, d \in \mathcal{C}_i$, we have

$$w(c, d) = 0.$$

Finally, for two different buckets $\mathcal{C}_i$ and $\mathcal{C}_{i'}$, and for two objects $c \in \mathcal{C}_i$ and $c' \in \mathcal{C}_{i'}$, we count the number of times c and c' are accessed within a volume distance of at most $m-1$.

If for some j and j' , we have $c = a_i^j$ and $c' = a_{i'}^{j'}$, or $c = b_i^j$ and $c' = b_{i'}^{j'}$, then the references to c and c' are within a volume distance of less than m if and only if they both occur in the sequence corresponding to the same vertex in T . The total number of such vertices is equal to the number of vertices u appearing in T such that $u_i = j$ and $u_{i'} = j'$, which is equal to $(m-1)(t-1)t^{m-3}$. Thus for these cases we have

$$w(a_i^j, a_{i'}^{j'}) = w(b_i^j, b_{i'}^{j'}) = (m-1)(t-1)t^{m-3}.$$

Alternatively, if for some j and j' , $c = a_i^j$ and $c' = b_{i'}^{j'}$, then one of the two cases may happen.

1. $i > i'$: Analogous to the above cases, the two accesses are within a volume distance of less than m if and only if they both occur

in the sequence corresponding to the same element in T . Thus in this case we have

$$w(a_i^j, b_{i'}^{j'}) = (m-1)(t-1)t^{m-3},$$

as before.

2. $i < i'$: (♣) In this case two accesses are within a volume distance of less than m if and only if the access to $b_{i'}^{j'}$ occurs in the sequence corresponding to a vertex u in T and the access to a_i^j occurs in the sequence corresponding to the vertex following u . We count the number of such consecutive vertices in T as follows. Let vertex v follow vertex u in T . We must have $u_{i'} = j'$ and $v_i = j$. Also, u and v must differ in exactly one component. Let it be the k th component. If $k = i$ or $k = i'$ then in each case, there are $t-1$ ways to choose the value for the differing component and the remaining components can be set in t^{m-3} ways. If $k \notin \{i, i'\}$, then after setting the values for the $m-4$ non-changing components, there are $t(t-1)$ ways to set the values for the differing component in the two vertices. Thus in this case, the total number of such pairs of accesses, and hence $w(a_i^j, b_{i'}^{j'})$ is equal to

$$2(t-1)t^{m-3} + (m-3)t(t-1)t^{m-4} = (m-1)(t-1)t^{m-3}.$$

Our case-by-case analysis above implies that for every two objects c and c' from two different buckets we have

$$w(c, c') = (m-1)(t-1)t^{m-3}.$$

Therefore, we can take t large enough to ensure that for every $m \geq 3$ the maximum matching solution matches x with y and matches every other object with an object in a different bucket.

Now, consider a data packing P which corresponds to a maximum matching in the graph constructed by our algorithm. We compute an upper bound on the number of cache hits the reference sequence incurs after we apply the packing P .

Since $(x, y) \in P$, and since there are always less than m accesses between two consecutive accesses to x and y , every access to these two objects will hit in the cache. The total number of these hits is equal to

$$4(m-1)(t-1)t^{m-1}.$$

For any object $c \notin \{x, y\}$, let $\mathcal{C}_i$ be the bucket containing c . We argued that P must pack c with an object in a different bucket, i.e., with an object $c' \in \mathcal{C}_{i'}$, for $i' \neq i$.

Every two consecutive accesses to c are separated by references to at least $2m-1$ distinct objects. Thus an access to c will hit in the cache only if c' is accessed during the $2m-1$ accesses preceding the access to c . One of the two following cases may happen.

1. Either $(c', c) \in \mathcal{A}_{i'} \times \mathcal{B}_i$ or $i > i'$ and $(c', c) \in \mathcal{A}_{i'} \times \mathcal{A}_i \cup \mathcal{B}_{i'} \times \mathcal{B}_i$: In these cases, the access to c hits in the cache only if the accesses to c and c' occur in the sequence corresponding to the same vertex in T . This can happen in $(m-1)(t-1)t^{m-3}$ ways.
2. Either $(c', c) \in \mathcal{B}_{i'} \times \mathcal{A}_i$ or $i < i'$ and $(c', c) \in \mathcal{A}_{i'} \times \mathcal{A}_i \cup \mathcal{B}_{i'} \times \mathcal{B}_i$: In these cases, the access to c hits in the cache only if the accesses to c and c' occur in the sequences corresponding to two consecutive elements in T . Via an analysis similar to (♣), this can happen in $(m-1)(t-1)t^{m-3}$ ways.

Summing up on all the $(m-1)t$ objects, there will be at most

$$(m-1)^2(t-1)t^{m-2}$$

cache hits happening for references to objects other than x and y . Thus the total number of hits is bounded by

$$H_P = 4(m-1)(t-1)t^{m-1} + (m-1)^2(t-1)t^{m-2}.$$

On the other hand, consider the packing P^* defined as

$$P^* = \left\{ (a_i^j, b_i^j) \mid \begin{array}{l} \forall i \in \{1, 2, \dots, m-1\} \\ \forall j \in \{1, 2, \dots, t\} \end{array} \right\} \cup \{(x, y)\}$$

All the references to x and y will hit in the cache. For every vertex u appearing in T , the sequence corresponding to u incurs exactly one cache miss. This miss happens at the reference to a_i^j , where i is the component in u whose value is different from the previous vertex in T . Thus under the packing P^* , the total number of cache hits is equal to

$$H_{P^*} = (2m+1)(m-1)(t-1)t^{m-1}.$$

Therefore, the performance guarantee of our algorithm is at most equal to

$$\frac{H_P}{H_{P^*}} = \frac{4}{2m+1} + \frac{m-1}{t} < \frac{2}{m},$$

where the last inequality holds if t is taken to be larger than $m(m^2 - 1)$.

This completes the proof that for an m -block cache and packing factor 2, our algorithm cannot achieve an approximation factor better than $m/2$. $\square$

5.1 An Iterative Packing Algorithm

Let us take a closer look at the reference sequence in the proof of our tightness result (Theorem 5.2). Why did our algorithm fail to find the optimal data packing scheme? It failed because at the stage where the algorithm was finding an approximate (or optimal) set packing solution, it was blind to the fact that packing x and y together would unlock the benefit from packing together, the objects that belong to the same bucket.

An iterative set packing algorithm can help us avoid this problem, while still retaining our approximation guarantees. Recall that an iterative p -set packing algorithm iteratively picks the heaviest set, until no sets remain that can be picked. We use this algorithm in the following way. After picking the heaviest remaining set S , we update the reference sequence to reflect the fact that all objects in S are packed in the same block. That is, for each $a \in S$, we substitute every reference to a in the reference sequence with a reference to the new object B_S which represents the single block containing S . Subsequently, the change in the reference sequence triggers an update in the proximity hypergraph. The naïve way to update this hypergraph is of course, to reconstruct it according to the new reference sequence, which results in a multiplicative factor of $O(n)$ in running time complexity, where n is the number of distinct objects in the reference sequence.

Even though this iterative algorithm achieves the same approximation factor as our previous algorithm, we believe that it will perform much better in practice. Optimizing the proximity hypergraph reconstruction phase can be an interesting future direction.

6. Related Work

Cache-conscious data placement has been extensively studied in the past. Rubin et al. categorized a collection of techniques including field reordering, field clustering (also called structure splitting and array regrouping), global variable positioning, coloring, linearization, clustering and custom memory allocation [31]. More recent techniques include cache partitioning [25] and array regrouping for parallel code [24]. Many techniques have been developed, including those based on access frequency [10, 19, 33], pairwise affinity [7, 26, 30], and group affinity [25, 37, 38, 40]. Rubin et al. gave a framework to profile and optimize using similar heuristics and search [31]. The frequency and affinity heuristics have also been used to improve code layout [15, 23, 28, 38]. Some of the

solutions are adopted in a commercial compiler [35, 39] and a tuning tool [24]. An even greater body of work including almost every research and industry tool for performance optimization and tuning is concerned with detecting and analyzing the problems of poor cache performance.

Given the importance of the problem in practice, it is of a broad significance to understand its complexity. The previous study by Petrank and Rawitz gave a thorough theoretical analysis including not only the complexity to optimize the memory layout but also the complexity to approximate the optimal solution [27]. The earlier analysis applies to cache conflicts. In this paper, we solve these problems for capacity misses. Our reduction uses very different techniques, including the separation of miss minimization from hit maximization and the use of reuse interval analysis. While our results complement the previous results to cover all types of cache misses incurred in a single cache, the complexity question remains open for optimal data placement for partitioned cache, which incurs an additional type of misses — coherence misses. The collection of theoretical techniques in this paper may help the further study on this subject.

7. Summary and Conclusion

Data packing is a challenging and important problem for today's block-based, highly associative caches. We formally defined this problem for an arbitrary number of cache blocks and packing factor. We showed, using a reduction from **3SAT** that even for a 5-block cache, an approximate data packing solution cannot be efficiently computed, unless $P = NP$. Furthermore, we showed by reducing from **2SAT-MIN-DEL**, that even for a 3-block cache, the existence of a constant factor approximation algorithm disproves a well-known conjecture.

In contrast, we showed that if, instead of minimizing cache misses, we aim at maximizing cache hits, the data packing problem will admit a constant factor approximation algorithm. While our approximation factor is linearly related to the number of cache blocks and the packing factor, our tightness analysis shows that such relation is inevitable for our algorithm.

Finally, in light of our tightness analysis, we proposed an iterative data packing algorithm that we believe performs better in practice.

Our theoretical study can help the data layout research in several ways. On the one hand, our hardness of approximation results enlightens a connection between the phase behavior and hardness of data packing. On the other hand, the techniques and ideas used in our cache hit maximization algorithms (Section 5) can prove helpful for general-purpose data layout optimization.

Acknowledgments This research was supported in part by NSF grants CCF-1116104, CNS-1319617, an IBM CAS Faculty Fellowship, and a research grant from Huawei. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the funding organizations.

I sincerely thank my advisor, Chen Ding, for introducing to me the earlier work on hardness of cache-conscious data placement, and Erez Petrank for his comments on preliminary results of this paper.

References

- [1] Cortex57 technical reference manual. 2012. Available at <http://www.arm.com/products/processors/cortex-a/cortex-a57-processor.php>.
- [2] Cortex72 technical reference manual. 2015. Available at <http://www.arm.com/products/processors/cortex-a/cortex-a72-processor.php>.

- [3] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the ACM Conference on Theory of Computing*, pages 305–314, 1987.
- [4] K. Andreev and H. Räcke. Balanced graph partitioning. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 120–124, 2004.
- [5] E. M. Arkin and R. Hassin. On local search for weighted k-set packing. In *Proceedings of the 5th Annual European Symposium on Algorithms*, pages 13–22, 1997.
- [6] T. N. Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Inf. Process. Lett.*, 42(3):153–159, 1992.
- [7] B. Calder, C. Krintz, S. John, and T. M. Austin. Cache-conscious data placement. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, 1998.
- [8] J. F. Cantin and M. D. Hill. Cache performance for SPEC CPU2000 benchmarks. <http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data>.
- [9] B. Chandra and M. M. Halldórsson. Greedy local improvement and weighted set packing approximation. *J. Algorithms*, 39(2):223–240, 2001.
- [10] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 1999.
- [11] U. Feige. Relations between average case complexity and approximation complexity. In *Proceedings of the 17th Annual IEEE Conference on Computational Complexity*, pages 534–543, 2002.
- [12] U. Feige and R. Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM J. Comput.*, 31(4):1090–1118, 2002.
- [13] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.
- [14] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 285–298, 1999.
- [15] N. C. Gloy and M. D. Smith. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, 1999.
- [16] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial & Applied Mathematics*, 9(4):551–570, 1961.
- [17] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *27th International Symposium on Computer Architecture*, pages 107–116, 2000.
- [18] E. Hazan, S. Safra, and O. Schwartz. On the complexity of approximating k-set packing. *Computational Complexity*, 15(1):20–39, 2006.
- [19] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 69–80, 2004.
- [20] C. Huneycutt, J. B. Fryman, and K. M. Mackenzie. Software caching using dynamic binary rewriting for embedded devices. In *31st International Conference on Parallel Processing*, pages 621–630, 2002.
- [21] S. Khot. On the power of unique 2-prover 1-round games. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing*, pages 767–775, 2002.
- [22] D. G. Kirkpatrick and P. Hell. On the completeness of a generalized matching problem. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 240–245, 1978.
- [23] P. Li, H. Luo, C. Ding, Z. Hu, and H. Ye. Code layout optimization for defensiveness and politeness in shared cache. In *Proceedings of the International Conference on Parallel Processing*, pages 151–161, 2014.
- [24] X. Liu, K. Sharma, and J. M. Mellor-Crummey. ArrayTool: a lightweight profiler to guide array regrouping. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 405–416, 2014.
- [25] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Soft-OLP: Improving hardware cache performance through software-controlled object-level partitioning. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 246–257, 2009.
- [26] N. McIntosh, S. Mannarswamy, and R. Hundt. Whole-program optimization of global variable layout. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 164–172, 2006.
- [27] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 101–112, 2002.
- [28] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, 1990.
- [29] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-way cache: Demand-based associativity via global replacement. In *32nd International Symposium on Computer Architecture*, pages 544–555, 2005.
- [30] R. M. Rabbah and K. V. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Transactions in Embedded Computing Systems*, 2(2), 2003.
- [31] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data layout optimizations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.
- [32] H. Saran and V. V. Vazirani. Finding k-cuts within twice the optimal. *SIAM Journal on Computing*, 24(1):101–108, 1995.
- [33] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, 1998.
- [34] S. Seo, J. Lee, and Z. Sura. Design and implementation of software-managed caches for multicores with local memory. In *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009)*, pages 55–66, 2009.
- [35] X. Shen, Y. Gao, C. Ding, and R. Archambault. Lightweight reference affinity analysis. In *Proceedings of the International Conference on Supercomputing*, pages 131–140, 2005.
- [36] K. O. Thabit. *Cache management by the compiler*. PhD thesis, Rice University, Houston, TX, 1982.
- [37] J. Yan, J. He, W. Chen, P.-C. Yew, and W. Zheng. ASLOP: A field-access affinity-based structure data layout optimizer. *SCIENCE CHINA Info. Sci.*, 54(9):1769–1783, 2011.
- [38] C. Zhang, C. Ding, M. Ogihara, Y. Zhong, and Y. Wu. A hierarchical model of data locality. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–29, 2006.
- [39] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral. Forma: A framework for safe automatic array reshaping. *ACM Transactions on Programming Languages and Systems*, 30(1):2, 2007.
- [40] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 255–266, 2004.