Efficient Approximations for Cache-Conscious Data Placement

Ali Ahmadi*
Sharif University of Technology
Tehran, Iran
ali.ahmadi.star27@gmail.com

Majid Daliri
University of Tehran
Tehran, Iran
daliri.majid@ut.ac.ir

Amir Kafshdar Goharshady†
The Hong Kong University of Science and Technology
Hong Kong, China
goharshady@cse.ust.hk

Andreas Pavlogiannis
Aarhus University
Aarhus, Denmark
pavlogiannis@cs.au.dk

Abstract

There is a huge and growing gap between the speed of accesses to data stored in main memory vs cache. Thus, cache misses account for a significant portion of runtime overhead in virtually every program and minimizing them has been an active research topic for decades. The primary and most classical formal model for this problem is that of Cache-conscious Data Placement (CDP): given a commutative cache with constant capacity k and a sequence Σ of accesses to data elements, the goal is to map each data element to a cache line such that the total number of cache misses over Σ is minimized. Note that we are considering an offline single-threaded setting in which Σ is known a priori. CDP has been widely studied since the 1990s. In POPL 2002, Petrank and Rawitz proved a notoriously strong hardness result: They showed that for every $k \geq 3$, CDP is not only NP-hard but also hard-to-approximate within any non-trivial factor unless P = NP. As such, all subsequent works gave up on theoretical improvements and instead focused on heuristic algorithms with no theoretical guarantees.

In this work, we present the first-ever positive theoretical result for CDP. The fundamental idea behind our approach is that real-world instances of the problem have specific structural properties that can be exploited to obtain efficient algorithms with strong approximation guarantees. Specifically, the access graphs corresponding to many real-world access sequences are sparse and tree-like. This was already well-known in the community but has only been used to design heuristics without guarantees. In contrast, we provide fixed-parameter tractable algorithms that provably approximate the optimal number of cache misses within any factor $1 + \epsilon$, assuming that the access graph of a specific degree d_ϵ is sparse, i.e. sparser real-world instances lead to tighter approximations. Our theoretical results are accompanied by an experimental evaluation in which our approach outperforms past heuristics over small caches with a handful of lines. However, the approach cannot currently handle large real-world caches and making it scalable in practice is a direction for future work.

CCS Concepts: • **Theory of computation → Parameterized complexity and exact algorithms**; • **Software and its engineering → Memory management**.

Keywords: cache management, parameterization, data placement, treewidth, cache misses, approximation

ACM Reference Format:
Ali Ahmadi, Majid Daliri, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2022. Efficient Approximations for Cache-Conscious Data Placement. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3519939.3523436

1 Introduction

Cache Misses. Most modern memory systems consist of a large but relatively slow main memory and one or more small but much faster cache levels. When a program wants to access a specific data item during its execution, the accessed data must first be present in the cache. Otherwise, it will be copied from the main memory to the cache, possibly causing the eviction of other data from the cache. This copying is called a "cache miss". Given the low speed of main memory, the back-and-forth copying between cache and main memory caused by cache misses is a significant

*A longer version is available at [1]. Authors are ordered alphabetically.
†Corresponding author

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 the author(s) 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.
PLDI '22, June 13–17, 2022, San Diego, CA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523436

contributor to runtime overheads in virtually all programs. Hence, minimizing cache misses has been a central problem in various communities, including programming languages [14, 19, 27, 35, 38, 50, 51], compilers [14, 36, 43, 46] and operating systems [11, 42, 49] for many decades.

Cache-conscious Data Placement (CDP). In this work, we focus on *Cache-conscious Data Placement* (CDP). CDP is arguably the most classical formulation for the problem of minimizing cache misses. It was first introduced in ASPLOS 1998 by Calder et al [14] and then further formalized by Petrank and Rawitz in POPL 2002 [38]. In this model, the memory system consists of two levels: a large main memory storing a set O of n distinct objects $o_1, o_2, \ldots, o_n$, and a small cache with k lines. Depending on the variant, each cache line can hold 1 or t objects. A *placement map* is a function $f : O \to \{1, 2, \ldots, k\}$ that maps each object to a cache line. When a placement map f is fixed and an access to an object o_i is requested, the system first checks to see whether o_i is already present in its corresponding cache line $f(o_i)$. If so, the access is successful. Otherwise, a cache miss happens and o_i must first be copied from the main memory to line $f(o_i)$ of the cache, potentially evicting another object that was already in this cache line. Only after this copying can the access go through. Given a sequence $\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_N \rangle \in O^N$ of accesses, CDP asks for a placement map f that minimizes cache misses over Σ.

Hardness of CDP. When considering the CDP problem, it is usually assumed that k and t are small constants and the complexity is studied with respect to the number of objects, i.e. n, and the length of the access sequence, i.e. N. In [38], Petrank and Rawitz showed that the CDP problem is NP-hard for any cache with more than two lines. They also showed that not only is the problem NP-hard, but it is also hard-to-approximate within any non-trivial factor $O(N^{1-\epsilon})$ unless P=NP. This became a notorious and well-known hardness result, causing all further works to focus on heuristics with no worst-case bounds on their approximation ratio. Some examples of this approach are [27, 28, 50, 51]. These heuristics try to identify and exploit affinities between data items to minimize cache misses.

Access Graphs and Their Sparsity. A recurring structure in the cache management literature is that of an *access graph* [12, 35, 46]. Simply put, an access graph is an undirected graph which has one vertex corresponding to each object $o_i \in O$ and an edge between two vertices if they appear consecutively in the access sequence Σ. Informally, the access graph models the simplest type of affinity between data items. Several previous works also consider extensions of access graphs to hypergraphs whose edges model affinities between more than two data items [19, 35, 46]. It is well-known that access graphs of real-world sequences are often sparse, opening the door to heuristics based on graph sparsity. Moreover, the optimal algorithm for data packing, which is another formalism for minimizing cache misses, is also based on the sparsity of access (hyper)graphs [19].

Our Focus. In this work, we consider the classical problem of Cache-conscious Data Placement (CDP) from an algorithmic and complexity point-of-view. Note that our setting is single-threaded and offline and we assume that the entire sequence Σ of accesses is given as part of the input. We focus on obtaining efficient algorithms that provably approximate the optimal number of cache misses within a constant multiplicative factor, assuming that the instance has sparse access (hyper)graphs. This assumption was already shown to hold for real-world instances in several previous works, such as [19]. We use the treewidth of the access (hyper)graphs as a measure of their sparsity.

Treewidth. Treewidth [10, 40, 41] is a well-known and oft-used graph sparsity parameter. Intuitively, the treewidth of a graph is a measure of its tree-likeness. Only trees and forests have a treewidth of 1 and if a graph's treewidth is w, then the graph can be decomposed into parts of size $w + 1$ that are connected to each other in a tree-like manner. See Section 2.2 for a more formal definition. The algorithmic importance of treewidth is due to the fact that many NP-hard graph problems are solvable in polynomial time over graphs of bounded treewidth [2, 6, 8, 9, 33]. Moreover, many families of graphs that appear in real-world contexts are shown to have small treewidth. This includes series-parallel and outer-planar graphs [7]. Control flow graphs of structured programs also have bounded treewidth [13, 15, 20, 34, 47], leading to faster program analysis and model checking algorithms [3, 16–18, 21–25, 30–32, 37, 39, 44]. Finally, access (hyper)graphs of many classical algorithms and programs are also shown to have small treewidth [19]. This is the family that is most relevant to the current work.

Intuition behind the Parameter. At first sight, treewidth of the access graph might come off as a surprising parameter. However, it is quite natural to expect this parameter to be small and this expectation was already confirmed by experiments in [19]. The intuitive reason behind this is that most real-world algorithms manipulate linear or tree-based data structures, such as arrays, vectors, linked lists, heaps, binary search trees and tries. Hence, the resulting access sequences consist of accesses to these tree-like structures and other helper variables which often have a short lifetime. So, the access graph inherits much of the sparsity and tree-likeness of the underlying data structures and the additional complexity introduced by temporary variables does not make it significantly denser. Treewidth is the classical parameter for capturing and formalizing such tree-like properties.

Our Contributions. We present the first positive theoretical results for the classical and notoriously-hard problem of Cache-conscious Data Placement (CDP). Our detailed results are as follows:

- *Approximation Scheme:* For every constant $\epsilon > 0$, we provide an efficient linear-time algorithm for CDP that is guaranteed to obtain a $(1+\epsilon)$-approximation of the optimal number of cache misses, assuming that the access graph of a specific degree d_ϵ has bounded treewidth. In other words, our scheme obtains tighter approximations for sparser instances.
- *Hardness Result:* We provide a stronger hardness result and show that CDP is NP-hard even when restricted to instances in which access hypergraphs of a fixed degree d have bounded treewidth. Intuitively, this suggests that both parameterization (sparsity) and approximation are needed in solving CDP. It is impossible to approximate CDP without a sparsity assumption as shown by [38]. On the other hand, our hardness result shows that it is also impossible to solve the problem exactly (without approximation) even when we assume that access hypergraphs of a fixed degree d are sparse.
- *Experimental Results:* We provide experimental results on the benchmarks of [19] and caches with 3–6 lines. On these small caches, our approach beats several well-known heuristics in the literature in terms of the number of cache misses.

Novelty. In summary, we provide the first positive theoretical result for CDP by a combination of approximation and parameterization. We also show a stronger hardness result that suggests both approximation and parameterization are probably necessary. Our algorithms are the first to provide provable bounds on the approximation ratio. To the best of our knowledge, graph sparsity parameters such as treewidth were not previously used in the context of CDP. We are also not aware of any other systems problem that is solved by applying both parameterization and approximation.

2 Preliminaries

In this section, we provide a formal definition of the CDP problem (mostly following [38]), as well as the necessary background from parameterized complexity.

2.1 Cache-Conscious Data Placement

Memory System. We consider a memory system consisting of a large main memory and a small cache with k *lines*. We also fix a set $O = \{o_1, o_2, \ldots, o_n\}$ of *objects (data items)*. We do not make any assumptions about the locations of objects in the main memory or its size.

Placement Map. A *placement map* is a function $f : O \to \{1, 2, \ldots, k\}$ that maps each object to a cache line.

Direct Mapping vs t-way Mapping. In *direct mapping*, each cache line can hold at most one data item at a time. In *t-way mapping*, each cache line can hold up to t objects. Our main focus is on the direct mapping case, but our approaches extend to t-way mapping as well.

Accesses and Cache Misses. Given a fixed placement map f, when an access to an object o_i is requested, o_i must first be present in cache line $f(o_i)$. If this is not the case, then a *cache miss* occurs and o_i is copied from the main memory to cache line $f(o_i)$. If this cache line is already full, another data item will be evicted from it. Note that if each cache line can hold more than one object, then we should also fix a replacement policy for each line. In this work, we assume that the replacement policy is LRU, i.e. the least recently used element is always evicted. This is because LRU is the most commonly-used policy in practice [51]. Our algorithms are also extensible to FIFO and ORP with minimal changes[1].

Access Sequence. An *access sequence* is simply a sequence $\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_N \rangle \in O^N$ of objects. Intuitively, Σ represents the order in which a program accesses the data items. We denote by $Misses_k^t(f, \Sigma)$ the number of cache misses that occur in a t-way cache with k lines if the placement map is f and the accesses are made according to Σ. We assume the cache is empty at the beginning and drop k when it is clear from the context. We also drop $t = 1$ in direct mapping.

Cache-conscious Data Placement (CDP). Given a set $O = \{o_1, \ldots, o_n\}$ of objects, an access sequence $\Sigma \in O^N$, and cache parameters t and k as input, the *Cache-conscious Data Placement* problem asks for an optimal placement map f^* that minimizes the number of cache misses. More formally, it asks for a placement map f^*, such that for any other placement map f, we have $Misses_k^t(f^*, \Sigma) \leq Misses_k^t(f, \Sigma)$.

Approximations. For an $\epsilon > 0$, we say that an algorithm is a $(1+\epsilon)$-approximation of CDP if given the same inputs, it always produces a placement map f such that $Misses_k^t(f, \Sigma) \leq (1+\epsilon) \cdot Misses_k^t(f^*, \Sigma)$.

Instances. An *instance* of the CDP problem is a tuple $I = (n, O, N, \Sigma, t, k)$ specifying all parts of the input.

2.2 Parameterization and Treewidth

Parameterized Complexity. The central idea in parameterized complexity is to analyze the runtime of an algorithm not only based on its input size n, but also based on another parameter p [29]. The parameter itself can be explicit, i.e. part of the input, or implicit, e.g. a structural property.

Fixed-parameter Tractability (FPT). A problem is called *Fixed-parameter Tractable* (FPT) wrt a parameter p, if there exists an algorithm that solves the problem in time $O(n^c \cdot g(p))$, where n is the input size, c is a constant that does not depend on either n or p and g is an arbitrary computable function [26, 29]. Intuitively, when a problem is FPT, the instances in which the parameter p is small are easy to solve and can be handled in polynomial time.

When dealing with a hard problem, such as CDP, the main challenge is to come up with a suitable parameter p, such

[1]The FIFO and ORP cases are removed since the space is limited and they do not provide new insights. We will publish them as a tech report.

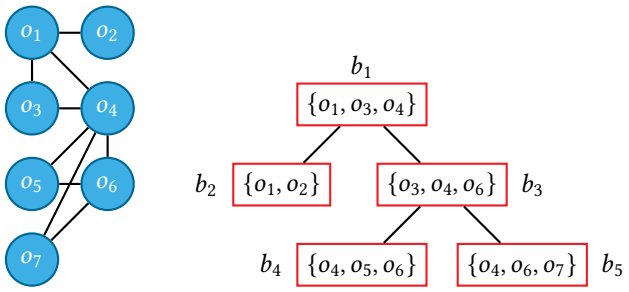

Figure 1. A graph $G = (V, E)$ (left) and a tree decomposition T of width 2 for G (right).

that (i) all or most real-world instances have a small p, and (ii) the problem becomes FPT wrt p. Finding such a parameter would effectively lead to efficient solutions for the real-world instances of the problem. We now define the parameter that will be used in this work, i.e. treewidth.

Graphs and Hypergraphs. A directed graph is a pair $G = (V, E)$ where V is a finite set of vertices and $E \subseteq V \times V$ is a finite set of edges. Each edge $e \in E$ is an ordered pair of vertices. An undirected graph is defined similarly, except that each edge e is a subset $\{u, v\} \subseteq V$. An ordered hypergraph is a pair $G = (V, E)$ where $E \subseteq V^+$, i.e. each hyperedge $e \in E$ is an ordered tuple of vertices in V. Similarly, in an unordered hypergraph, each edge e is simply a subset of vertices. The *base (hyper)graph* of a directed graph/ordered hypergraph is obtained by ignoring the order of vertices in each edge.

Tree Decompositions [26]. Consider an undirected / unordered (hyper)graph $G = (V, E)$. A *tree decomposition* of G is a rooted tree $T = (B, E_T, r)$ where:

(1) B is the set of nodes in the tree and E_T is the set of edges. We call each node in B a *bag* and $r \in B$ is the *root bag*.
(2) Each bag $b \in B$ has an associated subset $V_b \subseteq V$ of vertices. We reserve the word *vertex* for vertices of G.
(3) Each vertex appears in at least one bag, i.e. $\bigcup_{b \in B} V_b = V$.
(4) Each (hyper)edge appears in at least one bag. Formally, for every $e \in E$, there exists a bag $b \in B$, such that $e \subseteq V_b$. In other words, b contains all endpoints of e.
(5) Each vertex appears in a connected subtree of T. Equivalently, if a bag $b_3 \in B$ is on the unique path between the bags b_1 and b_2 in T, then $V_{b_3} \supseteq V_{b_1} \cap V_{b_2}$, i.e. if $v \in V$ appears in the two bags b_1 and b_2, then it must also appear on any bag b_3 that is on the unique path between them.

Note that tree decompositions do not distinguish between ordered/directed and unordered/undirected edges, i.e. a tree decomposition of an ordered/directed (hyper)graph is simply a tree decomposition of its base graph.

Example 1. *Figure 1 shows a graph G and one of its tree decompositions. Intuitively, in a tree decomposition the graph is broken into several small pieces (bags) that are connected to each other in a tree-like manner.*

Treewidth [26, 40]. The *width* of a tree composition is defined as the size of its largest bag minus 1, i.e. $w(T) := \max_{b \in B} |V_b| - 1$. The *treewidth* of a (hyper)graph G is the smallest width among all of its tree decompositions.

Cut Property [6, 26]. Consider a (hyper)graph G and a tree decomposition T of G and suppose that the vertices $v_1, v_2 \in V$ appear in bags $b_1, b_2 \in B$ respectively. Then every path from v_1 to v_2 in G has to pass through every bag b_3 that is on the path from b_1 to b_2 in T. This is called the *cut property* of tree decompositions.

Example 2. *The tree decomposition in Figure 1 has a width of 2 and is optimal. So, the treewidth of the graph G is also 2. Consider vertices $o_1 \in V_{b_2}$ and $o_7 \in V_{b_5}$. Since b_1 and b_3 are on the unique path from b_2 to b_5 in T, then any path that connects o_7 to o_1 in G has to intersect both of these bags. As an example, consider the path $\langle o_7, o_6, o_5, o_4, o_1 \rangle$. It intersects b_3 in both o_4 and o_6. Similarly, it intersects b_1 in both o_1 and o_4.*

Dynamic Programming. The cut property enables one to perform dynamic programming on low-treewidth graphs in a similar manner to trees. Intuitively, in dynamic programming approaches, each bag in a tree decomposition serves the same purpose as a vertex in a tree whose removal breaks the graph/tree down into several completely independent connected components. This can potentially lead to much faster algorithms, especially when the bags, and hence the treewidth, are small. See [6, 26] for some examples and a more detailed treatment.

Nice Tree Decompositions [26]. We say that a tree decomposition $T = (B, E_T)$ is *nice* if (i) the root bag and every leaf bag ℓ are empty, i.e. $V_r = V_\ell = \emptyset$, (ii) every bag has at most two children, (iii) if a bag b has a single child c, then b and c differ in exactly one vertex, i.e. $|V_b \Delta V_c| = 1$, and (iv) if a bag b has two children c_1 and c_2, then $V_b = V_{c_1} = V_{c_2}$. Every tree decomposition can be easily converted to a nice decomposition of the same width in linear time [26]. Nice decompositions help us in designing dynamic programming procedures in Section 3.3.

Computing Optimal Tree Decompositions. Given a graph G, computing its treewidth w and an optimal tree decomposition are FPT problems wrt w. Specifically, [5] provides a linear-time algorithm. Hence, we always assume that an optimal tree decomposition is given as part of the input.

3 An Efficient Parameterized Approximation Scheme for CDP

In this section, we first formalize the notion of access hypergraphs and then use it to provide an efficient parameterized approximation algorithm for CDP, consisting of two steps:

1. We reduce the problem of approximating CDP within any constant factor $(1 + \epsilon)$ to a special type of graph coloring on a sparsified subgraph of the access hypergraph (Section 3.2).

2. We provide a decomposition-based algorithm that efficiently solves the coloring problem on graphs that have bounded treewidth (Section 3.3).

Putting the two ingredients above together, we achieve an efficient solution for CDP which first turns CDP approximation into graph coloring and then efficiently colors the graph, assuming that it has bounded treewidth. Given that the bounded treewidth property holds for the access hypergraphs of many classical algorithms and programs [19], our approach can provide arbitrarily tight approximations of CDP in these cases. Throughout this section, we fix an input instance $I = (n, O, N, \Sigma, t, k)$.

3.1 Access Graphs and Access Hypergraphs

Access Graph. The access graph of a CDP instance $I = (n, O, N, \Sigma, t, k)$ is a directed graph $G = (O, E)$ in which every vertex is a data item and there is an edge between o_i and o_j if and only if o_i appears directly before o_j somewhere in the access sequence Σ. We do not add self-loops in G.

Example 3. *Consider the access sequence*

$$\Sigma = \langle o_1, o_2, o_1, o_4, o_5, o_3, o_3, o_1, o_2 \rangle.$$

Figure 2 shows the access graph of this sequence.

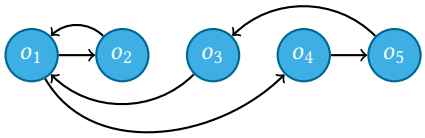

Figure 2. Access graph of Σ.

Access Hypergraphs. The access hypergraph of order d of the instance I is an ordered hypergraph $G_d = (O, E)$, in which there is an edge e_i corresponding to each access σ_i in $\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$. The edge e_i is of the form $\langle \sigma_j, \sigma_{j+1}, \ldots, \sigma_i \rangle$ in which $j \leq i$ is the largest index where at least one of the following holds:

1. $\langle \sigma_j, \sigma_{j+1}, \ldots, \sigma_i \rangle$ contains two accesses to σ_i.
2. $\langle \sigma_j, \sigma_{j+1}, \ldots, \sigma_i \rangle$ contains accesses to d distinct objects.
3. $j = 1$.

Informally, to form the edge corresponding to σ_i, we start from σ_i and move backwards in the access sequence until we either reach another access to σ_i or see d distinct data items or get to the beginning of the sequence.

Example 4. *Consider the same access sequence as in Example 3. Let $d = 3$. The access hypergraph of order 3 has the following edges:*

$$\begin{array}{lll} e_1 = \langle o_1 \rangle & e_2 = \langle o_1, o_2 \rangle & e_3 = \langle o_1, o_2, o_1 \rangle \\ e_4 = \langle o_2, o_1, o_4 \rangle & e_5 = \langle o_1, o_4, o_5 \rangle & e_6 = \langle o_4, o_5, o_3 \rangle \\ e_7 = \langle o_3, o_3 \rangle & e_8 = \langle o_5, o_3, o_3, o_1 \rangle & e_9 = \langle o_3, o_1, o_2 \rangle \end{array}$$

It is well-known that access hypergraphs are often sparse. In [19], the sparsity was formalized and it was shown that the access hypergraphs of many real-world algorithms and programs have bounded treewidth[2]. Thus, we will design FPT algorithms parameterized by the treewidth of a sparsified subgraph of the access hypergraphs.

3.2 Sparsification and Reduction to Graph Coloring

We now show how an approximation of the number of optimal cache misses in CDP can be obtained by reduction to a graph coloring problem over certain subgraphs of access hypergraphs.

Colorings. Consider a placement map $f : O \to \{1, 2, \ldots, k\}$. By definition, f assigns a cache line to every object $o_i \in O$. However, given that O is also the set of vertices in our access hypergraphs, one can equivalently think of f as a coloring of vertices in these graphs with k colors[3].

Direct Mapping. Let us first assume that we have a direct mapping instance, i.e. $t = 1$ and each cache line can hold only one object.

Sparsification. Consider the access hypergraph G_d of order d. Recall that G_d has an edge e_i corresponding to each access σ_i in Σ. We divide the edges of G_d in two groups: E_1 is the set of edges e_i that contain the vertex σ_i only once and E_2 is the set of edges e_i that contain σ_i twice. Let $\tilde{G}_d = (O, E_2)$ be the subgraph of G_d containing only the edges of the second kind. We call $\tilde{G}_d$ the *sparsified* access hypergraph of order d. Informally, $\tilde{G}_d$ keeps the edge corresponding to an access σ_i iff the number of other distinct data items seen since the last access to σ_i is less than d. The intuition is to focus on data items that are accessed regularly and whose placement in the memory really matters in the number of cache misses. These are elements that can likely cause capacity/conflict misses. In contrast, we would rather ignore elements that are accessed only once or rarely and cause a compulsory first-time cache miss anyway. In other words, if many distinct data items have been accessed since the last time we saw σ_i, then it is very likely that σ_i is already evicted from the cache and that the current access leads to a cache miss. Hence, we focus on minimizing the number of cache misses in accesses corresponding to E_2 only and assume all other accesses lead to cache misses. We will later see that discarding E_1 does not affect the optimal value too much, in the sense that the optimal solution to E_2 is always within a constant factor to the optimal solution overall. Hence, this leads to an approximation of the optimal number of cache misses within a constant multiplicative factor.

Canonical Hypergraphs. We say that an ordered hypergraph $G = (V, E)$ is *canonical* if every edge $e \in E$ is of

[2]The definition of access hypergraphs provided here is a bit different from [19] since we allow our hyperedges to include the same vertex more than once. However, this difference does not affect the treewidth.

[3]Adjacent vertices need not necessarily have different colors.

the form $\langle v_1, v_2, \ldots, v_m, v_1\rangle$ where $v_1 \notin \{v_2, \ldots, v_m\}$. In other words, every edge starts and ends with the same vertex and the start/end vertex does not appear anywhere else in the edge. Note that $\tilde{G}_d$ is canonical by definition.

Optimal Coloring. Consider a canonical hypergraph $G = (V, E)$ and a coloring function $f : V \to \{1, 2, \ldots, k\}$. We define $Cost(f, G)$ as the number of edges $e = \langle v_1, v_2, \ldots, v_m, v_1\rangle \in E$ such that $f(v_1) \in f(\{v_2, \ldots, v_m\})$, i.e. an edge contributes to the cost if it has an internal vertex with the same color as its start/end vertex. Such an edge is called a *missed edge*. Given a canonical G and a positive integer k as input, the *Optimal Coloring* problem asks for a coloring $\hat{f}$ with minimal cost, i.e. minimal number of missed edges.

The following lemma establishes a correspondence between missed edges in $\tilde{G}_d$ and cache misses in the CDP instance I.

Lemma 1. *Let f be a coloring of vertices in $\tilde{G}_d = (O, E_2)$ or equivalently a placement map for $I = (n, O, N, \Sigma, 1, k)$. An edge $e_i \in E_2$ is missed in the coloring f iff a cache miss occurs at its corresponding access σ_i with placement map f.*

Proof. Recall that e_i is of the form $\langle \sigma_j, \sigma_{j+1}, \ldots, \sigma_i\rangle$ and since $\tilde{G}_d$ is canonical we have $\sigma_i = \sigma_j$. If e_i is a missed edge, then there is some index $j + 1 \le l \le i - 1$ such that $f(\sigma_l) = f(\sigma_i)$. Hence, when an access to σ_l is made, the data item σ_i is evicted from the cache. As such, σ_i leads to a cache miss. Conversely, if no such l exists, then since $\sigma_j = \sigma_i$, this item has been moved to cache at time j and remained there until time i. So, there is no cache miss at σ_i. □

Corollary 1. $Misses_k(f^*, \Sigma) \le Misses_k(\hat{f}, \Sigma) \le Cost(\hat{f}, \tilde{G}_d) + |E_1|$.

Proof. Recall that f^* is the optimal placement map that minimizes the number of cache misses and $\hat{f}$ is the optimal coloring that minimizes the number of missed edges in $\tilde{G}_d$. Consider $\hat{f}$ as a placement map. Based on the lemma above, it causes exactly $Cost(\hat{f}, \tilde{G}_d)$ cache misses in accesses corresponding to E_2. It can also cause at most $|E_1|$ cache misses in accesses corresponding to E_1. □

This corollary allows us to bound the number of cache misses by solving the optimal coloring problem over the sparsified hypergraph $\tilde{G}_d$. We will later provide an algorithm for optimal coloring in Section 3.3. First, we provide a theorem showing that this approach leads to a constant approximation factor.

Theorem 1. *We have*

$$Misses_k(f^*, \Sigma) \le Misses_k(\hat{f}, \Sigma) \le \frac{d}{d-k} \cdot Misses_k(f^*, \Sigma).$$

Proof. The first inequality follows from the definition of f^*. Let M^* be the set of indices of accesses that lead to a cache miss if we use the optimal placement map f^* and $\hat{M}$ be the set of indices of accesses that lead to a cache miss when the optimal coloring $\hat{f}$ is used as the placement map. So, $|M^*| = Misses_k(f^*, \Sigma)$ and $|\hat{M}| = Misses_k(\hat{f}, \Sigma)$. Moreover, let L be the set of indices in Σ that correspond to edges in E_1 but did *not* lead to a cache miss in f^*. Note that we have

$$|M^*| \ge Cost(\hat{f}, \tilde{G}_d) + |E_1| - |L|. \tag{1}$$

To see this, let us count the number of misses caused by f^* in accesses corresponding to E_1 and E_2 separately. By definition of L, f^* causes $|E_1| - |L|$ cache misses in accesses of E_1. By definition of $\hat{f}$, we know that $Cost(f^*, \tilde{G}_d) \ge Cost(\hat{f}, \tilde{G}_d)$, so by Lemma 1, f^* causes at least $Cost(\hat{f}, \tilde{G}_d)$ cache misses in accesses corresponding to E_2. By combining Equation (1) and Corollary 1, we get

$$|\hat{M}| \le |M^*| + |L|. \tag{2}$$

So, it suffices to find a bound on $|L|$.

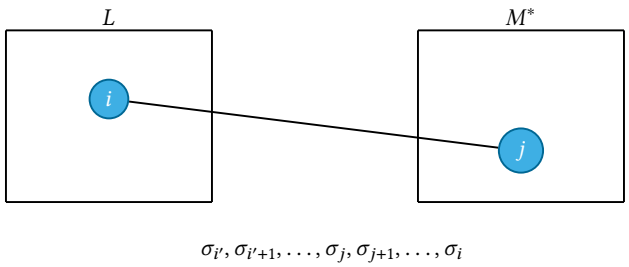

Figure 3. Construction of the bipartite graph $\mathcal{B}$. There is an edge from i in L to j in M^* iff in the access sequence Σ, σ_j is between σ_i and the previous access to the same element, i.e. $\sigma_{i'}$.

Let us form a bipartite graph $\mathcal{B}$ in which M^* serves as the set of vertices on one part and L as the set of vertices on the other part. Let $i \in L$, $j \in M^*$ and i' be the index of the previous access to σ_i, i.e. $i' = \max\{l < i | \sigma_l = \sigma_i\}$. Note that i' always exists, because if the first access to σ_i was at time i, then it would cause a cache miss with any placement map and hence i could not possibly be in L. We put an edge from the vertex i in L to the vertex j in M iff $i' < j < i$. See Figure 3. Note that the edges of $\mathcal{B}$ do not exactly correspond to cache misses. The only reason behind this construction is that counting the number of edges in two different ways enables us to bound $|L|$ in terms of $|M^*|$.

We now bound the number of edges of $\mathcal{B}$ in two ways. First, consider a vertex $i \in L$. The degree of i is the number of cache misses occurred between times $i' + 1$ and $i - 1$. Note that L only contains indices corresponding to E_1. Hence, at least d distinct data items were accessed in this period. At the end of time i', at most k of these items could potentially be in the cache. Thus, there are at least $d - k$ cache misses in this period, i.e. the degree of i is at least $d - k$, and the number of edges is at least $|L| \cdot (d - k)$.

Now consider a vertex $j \in M^*$. We prove that the degree of j is at most k. To get a contradiction, suppose that j has edges to $i_1, i_2, \ldots, i_k, i_{k+1} \in L$. Given that the range of f has k different values, by the pigeonhole principle there exist $a, b \in \{i_1, \ldots, i_{k+1}\}$ such that $f^*(\sigma_a) = f^*(\sigma_b) = f_0$. We know that $a' < j < a$ and $b' < j < b$. Without loss of generality, assume $a > b$. Since σ_a and σ_b are both mapped to f_0, σ_a was brought to cache line f_0 at time a' but was then evicted on or before time b. Hence, we have a cache miss at time a. This contradicts the definition of L. Therefore, the total number of edges is at most $|M^*| \cdot k$.

Putting the two bounds together, we get $|L| \le |M^*| \cdot \frac{k}{d-k}$. Combining this with (2), we have $|\hat{M}| \le |M^*| \cdot \frac{d}{d-k}$. □

Corollary 2. *For any $\epsilon > 0$, by applying the approach above using the sparsified access hypergraph $\tilde{G}_{d_\epsilon}$ of order $d_\epsilon := \lceil k + \frac{k}{\epsilon} \rceil$, we obtain a $(1+\epsilon)-$approximation of the optimal number of cache misses in a direct-mapped cache, i.e. $Misses_k(\hat{f}, \Sigma) \le (1+\epsilon) \cdot Misses_k(f^*, \Sigma)$.*

Extension to t-way Mapping. Extending the approach above to t-way mapping is quite straightforward and all steps go through naturally. Thus, we only present the differences. See [1, Appendix A] for a detailed treatment of the t-way mapping case.

Optimal t-way Coloring. In a canonical hypergraph $G = (V, E)$, we define $Cost^t(f, G)$ of a coloring function f as the number of edges $e = \langle v_1, v_2, \ldots, v_m, v_1 \rangle$ that have at least t *distinct* internal vertices with the same color as v_1. We call these edges *missed edges*. The *optimal $t-$way coloring* problem asks for a coloring $\hat{f}$ with minimal cost.

Lemma 1 and Corollary 1 apply to the t-way case with no changes and Theorem 1 sees only a minor change:

Theorem 2. *We have*

$$Misses_k^t(f^*, \Sigma) \le Misses_k^t(\hat{f}, \Sigma) \le \frac{d}{d - t \cdot k} \cdot Misses_k^t(f^*, \Sigma).$$

Proof. Every step is the same as in the proof of Theorem 1, except that the total cache size is now $t \cdot k$. Hence, the degree of each vertex in L is at least $d - t \cdot k$ and the degree of each vertex in M^* is at most $t \cdot k$. □

Corollary 3. *For any positive constant $\epsilon > 0$, by applying the approach above using the sparsified access hypergraph $\tilde{G}_{d_\epsilon}$ of order $d_\epsilon := \lceil t \cdot k + \frac{t \cdot k}{\epsilon} \rceil$, we obtain a $(1+\epsilon)-$approximation of the optimal number of cache misses in a t-way cache, i.e. $Misses_k^t(\hat{f}, \Sigma) \le (1+\epsilon) \cdot Misses_k^t(f^*, \Sigma)$.*

Remark. The proofs of the t-way results above, which are provided in detail in [1, Appendix A], are applicable even when the data items can have varying non-unit integer sizes. Hence, our approach is not limited to unit-sized objects.

Corollaries 2 and 3 show that we can get arbitrarily tight $(1+\epsilon)$-approximations of the optimal number of cache misses provided that we can solve the optimal (t-way) coloring problem on the sparsified access hypergraph of the right order and obtain the coloring/placement map $\hat{f}$. This is summarized in Algorithm 1. In Section 3.3, we provide a linear-time FPT algorithm for solving the optimal coloring and t-way coloring problems parameterized by treewidth. Hence, we can obtain a $(1+\epsilon)$-approximation of the number of cache misses whenever the sparsified access hypergraph $\tilde{G}_{d_\epsilon}$ is sparse and has bounded treewidth. It is also noteworthy that Theorems 1 and 2 and hence the $(1+\epsilon)$ factor are not tight. In practice, our approach may find much tighter approximations.

Algorithm 1 A $(1+\epsilon)$-approximation for CDP

```
1:  procedure CDP(n, O, N, Σ, t, k, ε)
2:      d ← ⌈t · k + (t·k)/ε⌉
3:      E_2 ← ∅
4:      for i = 1 to N do
5:          e_i ← ⟨σ_i⟩
6:          for j = i − 1 downto 1 do
7:              e_i ← ⟨σ_j⟩ + e_i
8:              if σ_j = σ_i then
9:                  E_2 ← E_2 ∪ {e_i}
10:                 break
11:             if |set(e_i)| > d then
12:                 break
13:     T = (B, E_T) ← NiceTreeDecomposition(O, E_2)
14:     return OptimalColoring(O, E_2, t, k, T)
```

3.3 A Decomposition-Based Algorithm for Optimal Coloring

In this section, we consider the problem of (t-way) optimal coloring, as defined in Section 3.2 and provide a linear-time FPT algorithm wrt treewidth for solving it, i.e. our algorithm can solve the problem in linear time if the input graph is sparse and has bounded treewidth.

Input. The input consists of two integers t and k, a *canonical* hypergraph $G = (V, E)$ with n vertices and N edges, each with at most d endpoints, and a *nice* tree decomposition $T = (B, E_T)$ of G with $O(n)$ bags and constant width w.

Output. The output is an optimal coloring function $\hat{f} : V \to \{1, 2, \ldots, k\}$ with minimal total cost.

Note that we are mostly focused on direct mapping, i.e. $t = 1$. However, our algorithm can handle any value of t. Moreover, we can assume that a tree decomposition of linear size is given as part of the input, since, as mentioned in Section 2.2, there are linear-time FPT algorithms for computing an optimal tree decomposition and making it nice.

Subtrees and Subgraphs. We say that a vertex v appears in a bag b, if $v \in V_b$. Similarly, an edge e appears in b if all

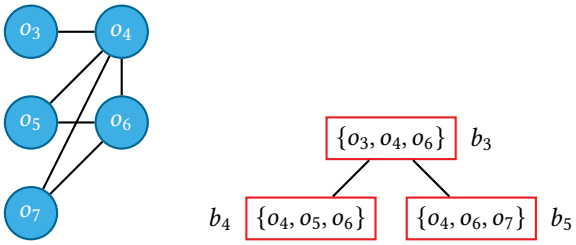

Figure 4. The subtree $T_{b_3}^{\downarrow}$ (right) and the subgraph $G_{b_3}^{\downarrow}$ (left) of the bag b_3 of Figure 1.

of its endpoints appear in b, i.e. $e \subseteq V_b$. We denote the set of all edges appearing in b by E_b. For a bag $b \in B$, we define its corresponding subtree $T_b^{\downarrow} = (B_b^{\downarrow}, E_{T_b^{\downarrow}})$ as the part of T that is rooted at b, i.e. including b and all of its descendants. The subgraph $G_b^{\downarrow}$ corresponding to b consists of all vertices and edges that appear in at least one bag in $T_b^{\downarrow}$, i.e. $G_b^{\downarrow} = \left(\bigcup_{b' \in B_b^{\downarrow}} V_{b'}, \bigcup_{b' \in B_b^{\downarrow}} E_{b'}\right)$.

Example 5. *Consider the graph and decomposition of Figure 1. Figure 4 shows the subtree and subgraph corresponding to the bag b_3.*

Partial Coloring. Let $b \in B$ be a bag. A *partial coloring* on b is simply a function $f_b : V_b \to \{1, 2, \ldots, k\}$ that assigns a color to each vertex in b. We denote the set of all $k^{|V_b|}$ possible partial colorings on b by C_b.

Our algorithm is a bottom-up dynamic programming on the nice tree decomposition T. There are two basic observations: (i) since every bag b is small and has size at most $w+1$, we can do a brute-force check of all partial colorings over b, and (ii) we can define subproblems on $G_b^{\downarrow}$ and its tree decomposition $T_b^{\downarrow}$ and use the solutions in these subproblems to solve the initial instances.

Dynamic Programming Variables. Based on the two observations above, for every bag $b \in B$ and partial coloring $f_b \in C_b$, the algorithm defines a dynamic programming variable $\mathsf{dp}[b, f_b]$ and initializes it to $+\infty$. Our goal is to compute values for the $\mathsf{dp}[\cdot, \cdot]$ in a bottom-up order such that the following invariant holds after we compute $\mathsf{dp}[b, f_b]$:

$$\mathsf{dp}[b, f_b] = \text{Minimal possible cost of a coloring of } G_b^{\downarrow} \text{ in which } V_b \text{ is colored according to } f_b \quad (\dagger)$$

In other words, we solve subproblems on $G_b^{\downarrow}$ corresponding to each possible partial coloring of b.

Computing dp Values. Our algorithm processes the bags of T in a bottom-up order and performs the following actions based on the type of the bag:

(1) Leaf Bags: Consider a leaf bag $\ell \in B$. Given that T is nice, we have $V_\ell = \emptyset$. Hence, C_ℓ contains only a single trivial coloring f_ℓ. Since there are no edges in $G_\ell^{\downarrow}$, the total cost would always be 0. Hence, the algorithm sets $\mathsf{dp}[\ell, f_\ell] = 0$.

(2) Bags with a Single Child: Suppose that $b \in B$ is a bag with a single child $c \in B$. Given that T is nice, we have $|V_b \,\Delta\, V_c| = 1$. The algorithm considers two cases:

(i) $V_b = V_c \cup \{v\}$, i.e. the bag b has one vertex v which does not appear in its child c: In this case, each partial coloring $f_b \in C_b$ induces a unique partial coloring $f_c := f_{b|V_c}$ on c. Hence, the minimal total cost in $G_c^{\downarrow}$ is $\mathsf{dp}[c, f_c]$ which is already computed in previous steps. The algorithm should compute $\mathsf{dp}[b, f_b]$, i.e. the minimal cost in $G_b^{\downarrow}$. The edges in $G_b^{\downarrow}$ can be divided in two sets: (a) edges that appear only in $G_b^{\downarrow}$ but not in $G_c^{\downarrow}$; and (b) edges that appear $G_c^{\downarrow}$. Note that every edge $e = \langle v_1, v_2, \ldots, v_m, v_1 \rangle$ in part (a) must have all of its endpoints in V_b. Hence, the partial coloring f_b fixes the colors of all v_i. So, the algorithm can simply iterate over the v_i's and check whether the edge e is missed. Moreover, the optimal cost (number of missed edges) in part (b) is given by $\mathsf{dp}[c, f_c]$. Thus, the algorithm sets $\mathsf{dp}[b, f_b] = \mathsf{dp}[c, f_c]$+ number of missed edges in (a).

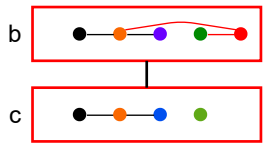

Figure 5. b has one vertex more than c. A coloring of b also colors c. New edges in $G_b^{\downarrow}$, i.e. part (a), are shown in red.

(ii) $V_c = V_b \cup \{v\}$, i.e. the child c has one vertex v which does not appear in its parent b: In this case, we have $G_b^{\downarrow} = G_c^{\downarrow}$. However, a partial coloring function $f_b \in C_b$ does not provide a color for the vertex v. Let $f_b[v \to i]$ be an extension of f_b that maps v to i. The algorithm sets

$$\mathsf{dp}[b, f_b] = \min_{i=1}^{k} \mathsf{dp}[c, f_b[v \to i]].$$

This is correct because $G_b^{\downarrow} = G_c^{\downarrow}$ and the only partial colorings in C_c that have no conflict with f_b are precisely those of the form $f_b[v \to i]$.

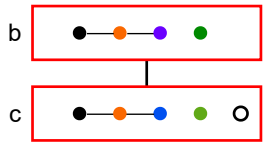

Figure 6. c has one vertex v more than its parent b. The coloring of f_b sets colors for all vertices of c except v. This vertex can have any color.

(3) Bags with Two Children: Consider a bag b with two children c_1 and c_2. Since T is nice, we have $V_b = V_{c_1} = V_{c_2}$ and $G_b^{\downarrow} = G_{c_1}^{\downarrow} \cup G_{c_2}^{\downarrow}$. So, when computing $\mathsf{dp}[b, f_b]$, we can use the same partial coloring function f_b for both c_1 and c_2 and then the number of missed edges in $G_b^{\downarrow}$ is equal to the number of missed edges in $G_{c_1}^{\downarrow}$ plus the number of missed edges in $G_{c_2}^{\downarrow}$ minus the number of missed edges that were counted in both. If an edge e is in both $G_{c_1}^{\downarrow}$ and $G_{c_2}^{\downarrow}$, then all of its endpoints must appear in both graphs. Using the last property of tree decompositions (see Section 2.2), we conclude that all of its endpoints have appeared in b and hence $e \in E_b = E_{c_1} = E_{c_2}$. Thus, the algorithm sets:

$$\mathsf{dp}[b, f_b] = \mathsf{dp}[c_1, f_b] + \mathsf{dp}[c_2, f_b] - \text{number of missed edges in } E_b$$

As before, the algorithm can check whether an edge $e \in E_b$ is missed because the partial coloring f_b provides the color information for all endpoints of e.

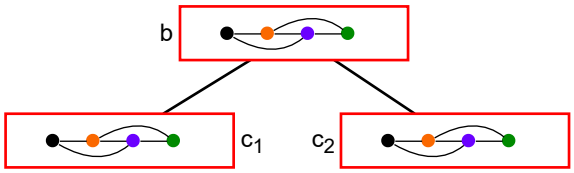

Figure 7. b has two children c_1 and c_2. A coloring of b will also color all vertices in V_{c_1} and V_{c_2}. Some edges are shared between $G_{c_1}^{\downarrow}$ and $G_{c_2}^{\downarrow}$. All such edges appear in b.

Computing the Final Answer. Since T is nice, we have $V_r = \emptyset$. So, there is only one possible partial coloring $\perp \in C_r$ for the root bag r. Moreover, we have $G_r^{\downarrow} = G$. So, the algorithm outputs $\mathsf{dp}[r, \perp]$ as the minimal number of missed edges. Algorithm 2 shows all steps of our method for obtaining the cost of the optimal coloring.

Finding the Optimal Coloring. The algorithm above obtains the minimal number of missed edges / minimal cost. As is common in dynamic programming, one can obtain the optimal coloring itself by simply keeping track of the partial colorings that led to the optimal $\mathsf{dp}[\cdot, \cdot]$ value at every step of the algorithm.

Theorem 3. *Given positive integer constants t and k, a canonical hypergraph G with n vertices whose edges have at most d endpoints, and a nice tree decomposition of G with $O(n)$ bags and width w, the algorithm above solves the t-way optimal coloring problem in total runtime $O(n \cdot k^{w+1} \cdot (k + d \cdot w^d))$.*

Proof. The correctness of the algorithm was argued in its presentation above. We focus on the runtime bound. There are $O(n)$ bags and the algorithm defines at most k^{w+1} different $\mathsf{dp}[\cdot, \cdot]$ variables at each bag b, since there are at most k^{w+1} partial colorings in C_b. Case (1) spends $O(1)$ time per variable and Case (2.ii) takes the minimum of k values in $O(k)$. In cases (2.i) and (3), all edges in the current bag should be checked to see if they are missed. There are at most $(w+1)^d$ such edges and checking each of them takes $O(d)$[4]. □

Algorithm 2 Parameterized algorithm for optimal coloring

```
1:  procedure OptimalColoring(V, E, t, k, T = (B, E_T))
2:      for b ∈ B in bottom-up order do
3:          for f_b : V_b → {1, 2, ..., k} do
4:              if b.children = ∅ then
5:                  dp[b, f_b] ← 0
6:              else if |b.children| = 1 then
7:                  c ← b.children[1]
8:                  if V_c ⊆ V_b then
9:                      v ← V_b \ V_c
10:                     f_c ← f_b|V_c
11:                     dp[b, f_b] ← dp[c, f_c]
12:                     for e ∈ E_b do
13:                         if v ∈ e ∧ is_missed(e, f_b) then
14:                             dp[b, f_b] ← dp[b, f_b] + 1
15:                 else if V_b ⊆ V_c then
16:                     v ← V_c \ V_b
17:                     dp[b, f_b] ← +∞
18:                     for i = 1 to k do
19:                         f_c ← f_b[v → i]
20:                         dp[b, f_b] ← min(dp[b, f_b], dp[c, f_c])
21:             else if |b.children| = 2 then
22:                 c_1, c_2 ← b.children
23:                 dp[b, f_b] ← dp[c_1, f_b] + dp[c_2, f_b]
24:                 for e ∈ E_b do
25:                     if is_missed(e, f_b) then
26:                         dp[b, f_b] ← dp[b, f_b] − 1
27:     r ← T.root
28:     return dp[r, ⊥]
```

Remark. The bound above is a theoretical worst-case bound and not tight. Our algorithm is indeed much faster in practice. Moreover, we can improve the runtime to $O(N \cdot k^{w+2})$ using a slightly different notion of nice tree decompositions. See [1, Appendix B] for details of theoretical improvements and Section 5 for experimental results.

Corollary 4. *When k, w and d are bounded, our algorithm solves the optimal coloring problem in linear time $O(n)$.*

We are now ready to present our main theorem:

Theorem 4. *For any $\epsilon > 0$, there exists an order d_ϵ, such that by applying our tree decompostion-based algorithm to the sparsified access hypergraph $\tilde{G}_{d_\epsilon}$, we obtain a linear-time*

[4]Without loss of generality, we can assume every edge in $\tilde{G}_d$ has exactly $d+1$ endpoints. Each edge has exactly d distinct vertices and removing repetitive internal vertices from the edge has no effect in our algorithm.

$(1+\epsilon)-$approximation of the optimal number of cache misses, as well as a placement map $\hat{f}$ such that

$$Misses_k^t(\hat{f}, \Sigma) \leq (1+\epsilon) \cdot Misses_k^t(f^*, \Sigma).$$

For direct mapping, we have $d_\epsilon = \lceil k + \frac{k}{\epsilon} \rceil$ and for t-way mapping, $d_\epsilon = \lceil t \cdot k + \frac{t \cdot k}{\epsilon} \rceil$.

Proof. Direct result of Corollaries 2, 3 and 4. □

4 Hardness of CDP in Bounded Treewidth

As proven in [38], it is impossible to approximate CDP within any non-trivial factor unless P=NP. In this section, we show that for every positive integer constant d, finding an *exact* solution to the CDP problem remains NP-hard even if the access hypergraph G_d of order d has constant treewidth. These two complementary hardness results show that both parameterization and approximation are necessary for our efficient solution in Section 3 and the problem remains NP-hard if only one of them is applied.

Theorem 5 (Hardness of CDP with Direct Mapping)**.** *For every positive integer constant d, the CDP problem with direct mapping is NP-hard even when limited to instances where the treewidth of G_d is bounded by a constant.*

Example 6. *Before providing a formal proof, let us illustrate the main ideas by an example. Our goal is to find a reduction from general CDP, which is NP-hard, to the special case of CDP in which the treewidth is bounded. Consider the access sequence of Example 3:*

$$\Sigma = \langle o_1, o_2, o_1, o_4, o_5, o_3, o_3, o_1, o_2 \rangle.$$

Suppose that we have a cache of size $k = 2$ and set $d = 2$ in the theorem above. In other words, we want to reduce our CDP instance $I = (n, O, N', \Sigma, 1, k) = (5, \{o_1, \ldots, o_5\}, 9, \Sigma, 1, 2)$ to another CDP instance $I' = (n', O', N', \Sigma', 1, k')$ such that the access graph of I' has small treewidth. We first introduce two new data elements (objects) τ_1 and τ_2 and set $O' = \{o_1, \ldots, o_5, \tau_1, \tau_2\}$. Intuitively, we want to take Σ and add $\langle \tau_1, \tau_2 \rangle$ between any two consecutive accesses, so that the treewidth of the access sequence becomes small. This leads to

$$\hat{\Sigma} = \langle \tau_1, \tau_2, o_1, \tau_1, \tau_2, o_2, \tau_1, \tau_2, o_1, \tau_1, \tau_2, o_4, \tau_1, \tau_2, o_5 \\ \tau_1, \tau_2, o_3, \tau_1, \tau_2, o_3, \tau_1, \tau_2, o_1, \tau_1, \tau_2, o_2, \tau_1, \tau_2 \rangle$$

Note that every access to any original data item o_i is now preceded and succeeded by the new elements τ_2 and τ_1. Ignoring edge directions and repetitions, this leads to an access graph that is almost bipartite, except for the edge between the new elements. See Figure 8. We can easily find a tree decomposition of constant width 2 for this access graph, as shown in Figure 9. We put a bag containing only the new elements as the root and add a child of the form $\{\tau_1, \tau_2, o_i\}$ for each o_i. It is easy to verify that this is a valid tree decomposition.

To have a reduction, we must be able to obtain the optimal number of cache misses in I from the optimal number of cache misses in I', but an optimal data placement for $\hat{\Sigma}$ might have no resemblance to its counterpart for Σ. So, we first increase our cache size by setting $k' = 4$, and then add a gadget that ensures each τ_i gets its own dedicated cache line. This ensures that exactly $2 = k$ cache lines remain for the o_i's and hence we can simulate the original instance. To achieve this property, we simply append many repetitions of $\langle \tau_1, \tau_2 \rangle$ to the end of $\hat{\Sigma}$, and define:

$$\Sigma' = \hat{\Sigma} \cdot \langle \tau_1, \tau_2 \rangle^{|\hat{\Sigma}+1|}.$$

In other words, Σ' is obtained by appending $|\hat{\Sigma} + 1|$ copies of $\langle \tau_1, \tau_2 \rangle$ to the end of $\hat{\Sigma}$. Note that in I', the new items τ_1 and τ_2 should be assigned to different cache lines. Otherwise, we will get $2 \cdot (\hat{\Sigma} + 1)$ cache misses in the second part of Σ' since every access to the new items will be a miss. In contrast, if they are assigned to the same cache line, we can get at most $|\Sigma'|$ cache misses in the first part and none in the second.

Now consider an optimal data placement for I' and suppose that it assigns τ_1 and some original object o_i to the same cache line. This means every access to o_i or τ_1 in $\hat{\Sigma}$ is a miss. We can modify our data placement and assign o_i to any other cache line that is not assigned to τ_1 or τ_2, and this will not increase the number of cache misses. In the worst case, every cache miss on o_i is preserved and every cache miss on τ_1 is replaced by a miss on another element that shares a cache line with o_i. Hence, there is an optimal data placement f' for I' in which τ_1 and τ_2 have their own dedicated cache lines. This means that the other elements must be put into $k' - 2 = k$ lines and hence I is simulated by I'. So, we can just count the number of cache misses on o_i's in I' and this gives us the optimal number of misses in I.

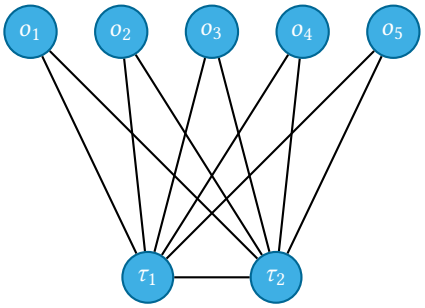

Figure 8. Base access graph of the sequence $\hat{\Sigma}$.

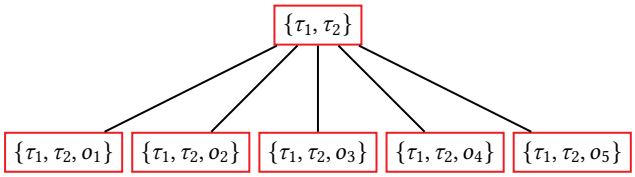

Figure 9. A tree decomposition of width 2 for the graph of Figure 8.

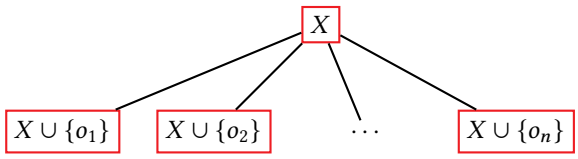

Figure 10. A decomposition of G'_d with constant width d.

Proof of Theorem 5. We provide a polynomial-time reduction from the general case of CDP to low-treewidth CDP. Since the former is NP-hard [38], then so is the latter. Let $I = (n, O, N, \Sigma, 1, k)$ be a CDP instance with direct mapping. We create a new CDP instance $I' = (n', O', N', \Sigma', 1, k')$ where:

- $n' = n + d$ and $O' = O \cup \{\tau_1, \tau_2, \ldots, \tau_d\}$, i.e. we add d new objects.
- $N' = d^2 \cdot N + d^2 + 2 \cdot d \cdot N + 2 \cdot d + N$ and the access sequence Σ' is of the following form:

$$X\ \sigma_1\ X\ \sigma_2\ X\ \ldots\ X\ \sigma_N\ X\ X^{d \cdot N + d + N + 1}$$

  where $X = \langle \tau_1, \tau_2, \ldots, \tau_d \rangle$. Intuitively, we add X at the beginning and end of Σ, as well as in between every two accesses. Finally, we add $d \cdot N + d + N + 1$ more copies of X to the end.
- $k' = k + d$, i.e. we add d new cache lines.

Let f' be an optimal placement function for I'. Note that for every $i \neq j$, we have $f'(\tau_i) \neq f'(\tau_j)$. This is because assigning τ_i and τ_j to the same cache line will cause at least $d \cdot N + d + N + 1$ cache misses in the final part of Σ', i.e. in $X^{d \cdot N + d + N + 1}$, whereas any placement that assigns different cache lines to each of the τ_l's leads to no cache misses in this part. The length of the rest of the sequence is $d \cdot N + d + N$ which is a natural upper-bound on the number of possible cache misses. Next, we argue that there is an optimal f' that does not assign any τ_i and o_j to the same cache line. Suppose that $f(\tau_i) = f(o_j)$. Then every access to o_j at any time a is a cache miss, since $f(o_j)$ contains τ_i. Similarly, the access to τ_i at time $a + i$ is also a cache miss. We now change $f(o_j)$ arbitrarily to some other value q that is not shared with any τ_l. It is easy to verify that this cannot increase the number of cache misses. In the worst case, the misses on o_j remain and the misses on τ_i are replaced by misses on the first other access that is mapped to q. By repeating this process, we will obtain an optimal f' that uses d of the cache lines for $\{\tau_1, \ldots, \tau_d\}$ and the other k lines for O. Hence, $f^* = f'_{|O}$ is an optimal solution for I. This completes the reduction. Figure 10 is a decomposition of this graph with width d. □

Theorem 6 (Hardness of CDP with t-way Mapping)**.** *For all positive integer constants d and t, the CDP problem with t-way mapping is NP-hard even when limited to instances where the treewidth of G_d is bounded by a constant.*

Proof. Section 4.3 of [38] provides a construction that, by introducing new data items and polynomially increasing the instance size, simulates a direct-mapping cache by a $t-$way mapping cache. The construction in [38] uses a constant number of extra data elements and does not blow up the treewidth of G_d. We can then apply Theorem 5. □

5 Experimental Results

In this section we report on an implementation and experimental evaluation of our algorithm for CDP.

Implementation. We implemented our approach, i.e. the algorithm of Section 3.3 for direct-mapped caches with the optimizations of [1, Appendix B], in C++ and used the LibTW library [48] for computing optimal tree decompositions.

Machine. All results were obtained on an Ubuntu 20.04 machine using a single thread of an Intel Xeon E3-1220 v2 Processor (3.1 GHz, 8M Cache) with 32 GB of RAM.

Benchmarks. We used the benchmarks of [19] for obtaining experimental results. These benchmarks contain access sequences Σ that are generated from a wide variety of classical algorithms including in linear algebra, sorting, divide-and-conquer, dynamic programming and string matching. In [19], they were introduced as benchmarks for the problem of data packing, which is another formalism of minimizing cache misses. Given that both data packing and CDP have the same input format, i.e. an access sequence of a program, we can simply repurpose the benchmarks of [19] for our use-case. Each benchmark corresponds to a classical algorithm, e.g. Gram-Schmidt or Heap Sort, and can generate access sequences of various (arbitrarily long) lengths. See [19] and its artifact for a complete list of benchmarks and other details.

Test Cases. Recall that a direct-mapping instance is a tuple $I = (n, O, N, \Sigma, 1, k)$. Our algorithm also needs an extra parameter d, i.e. the degree of the access hypergraph. We call the tuple (n, O, N, Σ, k, d) a *test case*. In our experiments, we set a time limit of 5 minutes per test case for our algorithm and, for each benchmark, each cache size $3 \leq k \leq 6$, and each hypergraph degree $k < d < 15$, generated all the test cases that our algorithm could handle in this time limit. This led to a total of 12,085 test cases, corresponding to 1,633 distinct instances. Our longest access sequence in our instances has 12,917 accesses. Note that the cache sizes considered here are much smaller than those in the real world. Our algorithm is hence not suitable for practical cache management but can instead be used for limit studies and profiling (See Section 6). Similarly, note that we assume the entire sequence Σ of accesses is given as part of the input and are solving the single-threaded offline case of the problem.

Sparsity of Instances. The fact that access graphs and access hypergraphs are sparse is quite well-known. In [19], it was shown that the access hypergraphs of most classical

algorithms have bounded treewidth. However, in contrast to previous methods, our algorithm does not depend on the access hypergraph G_d itself, but only on a sparsified subgraph $\tilde{G}_d$. See Section 3.2. This means that we work with a much sparser graph. In our experiments, the average ratio of the number of edges in $\tilde{G}_d$ to the number of edges in G_d was 47.22%. So, our sparsification has significant impact, leading to graphs that have less than half as many edges as the widely-used access hypergraphs. Moreover, they have a treewidth of at most 14. Figure 11 provides a histogram of the treewidths of $\tilde{G}_d$ in our test cases.

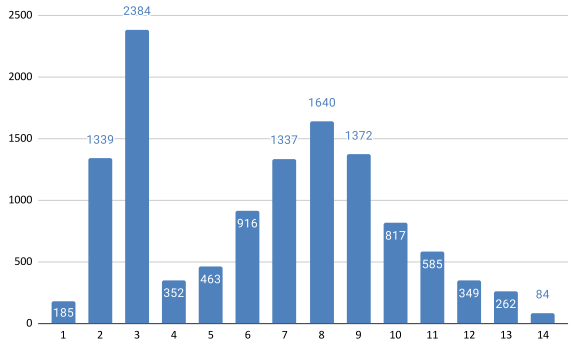

Figure 11. Treewidths of our test cases. The x axis is the treewidth of the sparsified access hypergraph $\tilde{G}_d$ and the y access is the number of cases.

Baselines. We compare our algorithm against several well-known heuristics in the literature.

- CKJA: This is the algorithm presented in [14], when cache-conscious data placement was first defined. It is a classic and has since been extensively studied.
- BB: This algorithm was presented in [4] and provides a graph-theoretic approach that aims to totally prevent the so-called "conflict misses" if possible.
- SCE: This approach aims to minimize cache misses using a coloring-based heuristic. It was presented in [45].

Experimental Results. Table 1 provides a summary of the number of instances where our approach outperformed the baseline heuristics. Overall, our algorithm beats CKJA in **85%** of instances, BB in **84%** and SCE in **88%**. Figure 12 provides a detailed comparison between our algorithm and the baselines above. In this figure, each red dot corresponds to one instance. The dot's x coordinate is the number of cache misses obtained by our algorithm and its y coordinate is the number of cache misses of the other method. The $x = y$ line is shown in blue. Hence, a red dot above the line corresponds to an instance in which our algorithm performed better than the other approach, and a red dot below the blue line signifies that the other approach performed better.

Table 1. Comparison of our algorithm with the baselines. The total number of benchmarks instances is 1633. Each cell contains the number of instances in which our algorithm outperformed the base line (left) and the number of instances in which the baseline had fewer misses (right).

	CKJA	BB	SCE
Our Algorithm	238 / 1395	260 / 1373	192 / 1441

Lower-bounds. A major theoretical advantage of our approach is that, for the first time, it provides constant multiplicative approximation ratio guarantees. Specifically, we can use the guaranteed ratio in Theorem 1 to obtain a lower-bound ℓ on the optimal number of cache misses, i.e. we are guaranteed to have at least ℓ cache misses no matter which placement function is used. These lower-bounds are shown in Figure 13. As before, there is a green dot corresponding to each instance. The green dot's x coordinate is the number of cache misses obtained by our algorithm, whereas its y coordinate is the guaranteed lower-bound ℓ. As expected, all green dots are below the $x = y$ line.

Summary. Our experimental results show that our novel approach manages a better utilization of the cache compared to previous heuristics, leading to improved cache performance in the vast majority of the benchmarks. Moreover, the performance gap increases as we go to more demanding benchmarks, indicated by the widening distribution of data points on the right-end side of the charts in Figure 12. Our approach is the first to provide theoretical guarantees of approximation within a constant ratio. Although our running time is generally larger than the heuristics, it is many orders of magnitude faster than a purely exhaustive search, which is the only other known approach so far that offers any non-trivial guarantees of optimality. Performing exhaustive search on our benchmark instances will take more than 10^{500} years per instance. This matches the intuition provided by the notorious hardness-of-approximation result in [38]. Our parameterized approach overcomes this hardness of approximation and solves instances that have thousands or even tens of thousands of accesses. This being said, given that our runtime depends exponentially on the cache size, we can only handle small caches and our approach does not scale to real-world cache sizes. Finally, our lower bounds can be used in limit studies of heuristics, in order to characterize their performance not against another approach, but compared to the best theoretically-possible performance.

6 Further Discussion and Related Works

Limitations. The primary limitation of our approach is that it is only applicable in the offline setting in which the entire access sequence is known a priori. Note that all previous hardness results were also for the same offline case. Our

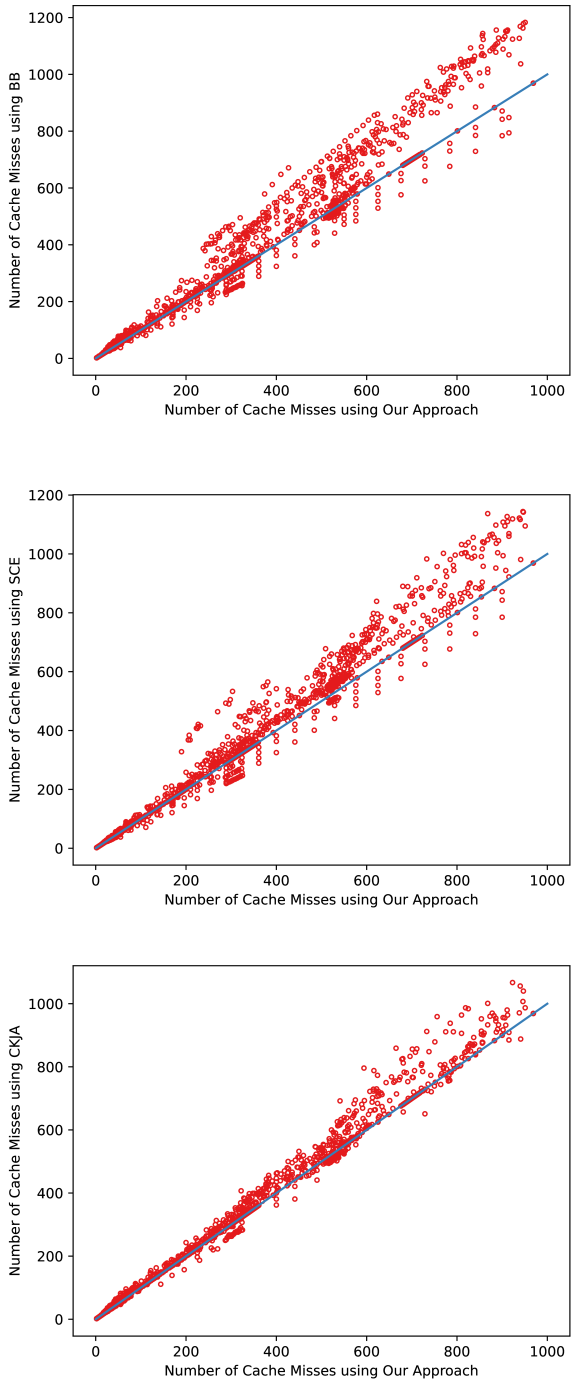

Figure 12. Performance of our algorithm vs BB (top), SCE (middle), and CKJA (bottom).

experimental results demonstrate that our approach leads to fewer misses than previous heuristics in the literature. However, it can currently handle only small caches with a handful of lines. More specifically, we provide algorithms

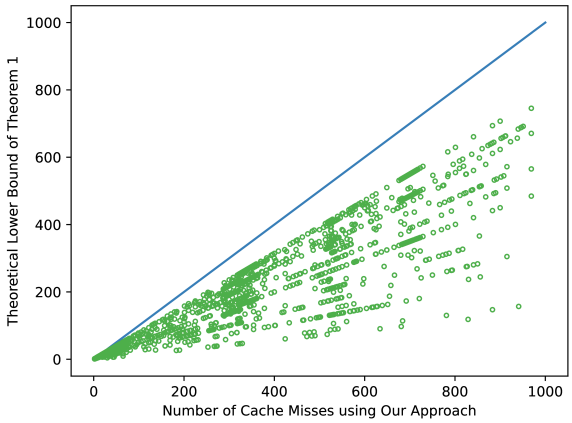

Figure 13. Number of cache misses obtained by our algorithm vs the theoretical lower bounds of Theorem 1.

with runtimes of either $O(N \cdot k^{w+2})$ or $O(n \cdot k^{w+1} \cdot (k+d \cdot w^d))$, in which k is the cache size, d is the order of the access hypergraph and w is the treewidth. Thus, while we overcome the hardness-of-approximation and provide the first polynomial-time algorithms with approximation-ratio guarantees, more improvement is needed to handle larger instances. Our results strongly indicate that solving real-world instances of CDP, within a provably-bounded approximation factor, is likely within reach and not as hard as previously thought. Moreover, they show that while the general case of the problem is NP-hard and hard-to-approximate, this is not the case for the sparse instances that are often encountered in practice. Another limitation is that our problem only models the single-threaded case and no parallelism is allowed in accesses to the cache.

Offline vs Online. While it is more desirable to minimize cache misses in an online setting, where the entire access sequence Σ is not known in advance, the problem is often studied in offline mode and Σ is assumed to be part of the input. This applies not only to this work but also all previous theoretical results on both data packing [19, 35] and CDP [38]. It is partly because the offline variants are already too hard, i.e. NP-hard and hard-to-approximate. On the other hand, solving the offline version is also useful in the following two cases (taken from [19]):

- *Limit studies:* To test the performance of a compiler for data placement, various inputs are generated as benchmarks, and the baseline comparison of the performance is performed against the best-known offline algorithm [38]. Hence, an almost-optimal algorithm with guaranteed approximation ratio for the offline case is needed.
- *Profiling:* Programs usually have similar memory access behaviors over different inputs [38]. Hence, an

effective approach for online cache management is to consider several representative inputs, run an almost-optimal offline algorithm for profiling, and then synthesize an answer to the online case from the offline solutions [14, 38]. Specifically, the traditional approach of [14] for online CDP is to assign a cost to each pair (o_i, o_j) of elements which roughly correlates with the number of extra cache misses that will be caused by assigning both o_i and o_j to the same cache line. This cost is always approximated using various profiling techniques. For example, we can run a program over thousands of random inputs and solve the offline variant of CDP for each run. Then, the cost we assign to (o_i, o_j) should be inversely correlated with the number of test cases in which o_i and o_j were put in the same cache line. The online algorithm will then simply work greedily and, upon the first access to an element o_i, assign it to a cache line that minimizes its cost. Alternatively, we can devise a supervised ML algorithm for the online case in which the outputs of the optimal offline algorithm are used as the training set.

As such, the offline case considered in this work, while not leading to practical algorithms that can be directly used for cache management, is still useful both theoretically and for the applications above.

Paging. Paging is a related well-studied problem, in which objects (or blocks) are not assigned to any specific cache line. This is equivalent to having a cache with a single line that can hold up to k objects. The goal is to find an optimal replacement policy that minimizes the total number of cache misses [12], i.e. to find the optimal policy for choosing which object should be evacuated each time new data is brought into the cache. Common replacement policies include FIFO, which evicts the object that has been in the cache for the longest, and LRU, which evicts the least-recently used/accessed object [19, 35, 51]. In the offline case, where the sequence Σ of accesses is known in advance, the Optimal Replacement Policy (ORP) is to evict the object whose next access is furthest in the future [12].

Data Packing. Data packing is another formulation of the problem of minimizing cache misses. In this case, the objects are not assigned to specific cache lines. Instead, they are "packed" into blocks of a fixed size and the cache can hold a fixed number of blocks. The goal is to find a packing that minimizes the total number of cache misses over a given access sequence Σ [46]. Similar to CDP, data packing is also NP-hard and hard-to-approximate within any non-trivial factor unless P=NP [35].

Comparison with [19]. The work [19] provides an algorithm for the problem of data packing using a parameterization by the treewidth of the access hypergraphs. The parameter we use in this work is similar, but not exactly the same. Specifically, we consider the treewidth of a sparsified subgraph of the access hypergraphs (Section 3.2). This sparsification is a key part of our theoretical contribution and necessary for obtaining a constant-ratio approximation. Additionally, the two works also differ in the following ways:

- *Modeling of the Cache:* [19] considers the problem of Data Packing (DP), whereas we study Cache-conscious Data Placement (CDP). As mentioned above, DP and CDP model the cache differently. In CDP, each data item is mapped to a specific cache line, whereas in DP, the items do not have a fixed position in the cache but are instead grouped (packed) together to form blocks.
- *Hardness and Parameterized Complexity:* While both CDP and DP are NP-hard and hard-to-approximate, the DP problem of [19] becomes fixed-parameter tractable and admits a polynomial-time algorithm when the treewidth is bounded. In contrast, our problem remains NP-hard even when limited to graphs of constant treewidth (Section 4) and can only be approximated. Hence, we are considering a strictly harder problem in terms of parameterized complexity and the techniques of [19] are not applicable to our setting.
- *Solution Concepts:* Both our solution and that of [19] reduce cache management problems to variants of graph coloring. In [19], the *number of vertices of any given color* is bounded, whereas in our case the *number of colors* is at most the cache size k.

7 Conclusion

We studied CDP, which is a standard and classical problem in memory management. As previous works have either formal and strong theoretical hardness results, or heuristics with no guarantees of optimality, this work is the first to present formal *positive* results. Particularly, we showed real-world instances of CDP admit efficient approximations within a constant ratio $(1+\epsilon)$ based on sparsification and parameterization by treewidth. Notably, our results differ from standard algorithmic approaches in which treewidth suffices to make the problem tractable and we show the problem *remains NP-hard* even with bounded treewidth, and only approximations are possible.

Interesting directions of future work include studying the existence of other parameters that allow for an efficient algorithm to solve CDP *exactly*, designing heuristics on top of our treewidth-based algorithm to improve its performance, and on the more practical side, incorporating our algorithm in data placement processes of mainstream compilers.

Acknowledgments

We are grateful to the anonymous reviewers whose suggestions significantly improved this paper. The research was supported by the HKUST–Kaisa Joint Institute Grant HKJRI3A055. A. Ahmadi and M. Daliri were interns at HKUST.

References

[1] Ali Ahmadi, Majid Daliri, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2022. Efficient Approximations for Cache-conscious Data Placement. https://hal.archives-ouvertes.fr/hal-03616652/

[2] Mohsen Alambardar, Amir Goharshady, Mohammad Reza Hooshmandasl, and Ali Shakiba. 2021. Optimal Mining: Maximizing Bitcoin Miners' Revenues. (2021). https://hal.archives-ouvertes.fr/hal-03232783

[3] Ali Asadi, Krishnendu Chatterjee, Amir Goharshady, Kiarash Mohammadi, and Andreas Pavlogiannis. 2020. Faster algorithms for quantitative analysis of MCs and MDPs with small treewidth. In *ATVA*. 253–270.

[4] Mirza Beg and Peter Van Beek. 2010. A graph theoretic approach to cache-conscious placement of data for direct mapped caches. In *ISMM*. 113–120.

[5] Hans Bodlaender. 1996. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing* 25, 6 (1996), 1305–1317.

[6] Hans Bodlaender. 1997. Treewidth: Algorithmic techniques and results. In *MFCS*. 19–36.

[7] Hans Bodlaender. 1998. A Partial k-Arboretum of Graphs with Bounded Treewidth. *Theor. Comput. Sci.* 209, 1-2 (1998), 1–45.

[8] Hans L Bodlaender. 1988. Dynamic programming on graphs with bounded treewidth. In *ICALP*. 105–118.

[9] Hans L Bodlaender. 1994. A tourist guide through treewidth. *Acta cybernetica* 11, 1-2 (1994), 1.

[10] Hans L Bodlaender. 2005. Discovering treewidth. In *SOFSEM*. 1–16.

[11] Hendrik Borghorst and Olaf Spinczyk. 2019. CyPhOS - A Component-Based Cache-Aware Multi-core Operating System. In *ARCS*. 171–182.

[12] Allan Borodin, Sandy Irani, Prabhakar Raghavan, and Baruch Schieber. 1995. Competitive Paging with Locality of Reference. *J. Comput. Syst. Sci.* 50, 2 (1995), 244–258.

[13] Bernd Burgstaller, Johann Blieberger, and Bernhard Scholz. 2004. On the tree width of Ada programs. In *ADA*. 78–90.

[14] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-Conscious Data Placement. In *ASPLOS*. 139–149.

[15] Krishnendu Chatterjee, Amir Goharshady, and Ehsan Goharshady. 2019. The treewidth of smart contracts. In *SAC*. 400–408.

[16] Krishnendu Chatterjee, Amir Goharshady, Prateesh Goyal, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2019. Faster algorithms for dynamic algebraic queries in basic RSMs with constant treewidth. *TOPLAS* 41, 4 (2019), 1–46.

[17] Krishnendu Chatterjee, Amir Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2016. Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In *POPL*.

[18] Krishnendu Chatterjee, Amir Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2020. Optimal and perfectly parallel algorithms for on-demand data-flow analysis. In *ESOP*. 112–140.

[19] Krishnendu Chatterjee, Amir Goharshady, Nastaran Okati, and Andreas Pavlogiannis. 2019. Efficient parameterized algorithms for data packing. In *POPL*. 53:1–53:28.

[20] Krishnendu Chatterjee, Amir Goharshady, and Andreas Pavlogiannis. 2017. JTDec: A tool for tree decompositions in soot. In *ATVA*. 59–66.

[21] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Amir Goharshady, and Andreas Pavlogiannis. 2018. Algorithms for algebraic path properties in concurrent systems of constant treewidth components. *TOPLAS* 40, 3 (2018), 1–43.

[22] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2015. Faster algorithms for quantitative verification in constant treewidth graphs. In *CAV*. 140–157.

[23] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2016. Optimal reachability and a space-time tradeoff for distance queries in constant-treewidth graphs. In *ESA*, Vol. 57.

[24] Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2021. Quantitative Verification on Product Graphs of Small Treewidth. In *FSTTCS*.

[25] Krishnendu Chatterjee and Jakub Łącki. 2013. Faster algorithms for Markov decision processes with low treewidth. In *CAV*. 543–558.

[26] Marek Cygan, Fedor Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. 2015. *Parameterized algorithms*. Springer.

[27] Chen Ding and Ken Kennedy. 1999. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In *PLDI*. 229–241.

[28] Wei Ding and Mahmut Kandemir. 2014. CApRI: CAche-conscious data reordering for irregular codes. In *SIGMETRICS*. 477–489.

[29] Rodney Downey and Michael Fellows. 2012. *Parameterized complexity*. Springer.

[30] John Fearnley and Sven Schewe. 2012. Time and parallelizability results for parity games with bounded treewidth. In *ICALP*. 189–200.

[31] Andrea Ferrara, Guoqiang Pan, and Moshe Y Vardi. 2005. Treewidth in verification: Local vs. global. In *LPAR*. 489–503.

[32] Amir Goharshady. 2020. *Parameterized and algebro-geometric advances in static program analysis*. Ph.D. Dissertation. Institute of Science and Technology Austria.

[33] Amir Goharshady and Fatemeh Mohammadi. 2020. An efficient algorithm for computing network reliability in small treewidth. *Reliability Engineering & System Safety* 193 (2020), 106665.

[34] Jens Gustedt, Ole A Mæhle, and Jan Arne Telle. 2002. The treewidth of Java programs. In *ALENEX*. 86–97.

[35] Rahman Lavaee. 2016. The hardness of data packing. In *POPL*. 232–242.

[36] Abraham Mendlson, Shlomit Pinter, and Ruth Shtokhamer. 1994. Compile Time Instruction Cache Optimizations. In *CC*. 404–418.

[37] Jan Obdržálek. 2003. Fast mu-calculus model checking when tree-width is bounded. In *CAV*. 80–92.

[38] Erez Petrank and Dror Rawitz. 2002. The hardness of cache conscious data placement. In *POPL*. 101–112.

[39] Leon R Planken, Mathijs M de Weerdt, and Roman PJ van der Krogt. 2012. Computing all-pairs shortest paths by leveraging low treewidth. *JAIR* 43 (2012), 353–388.

[40] Neil Robertson and Paul Seymour. 1984. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B* 36, 1 (1984), 49–64.

[41] Neil Robertson and Paul D. Seymour. 1986. Graph minors. II. Algorithmic aspects of tree-width. *Journal of algorithms* 7, 3 (1986), 309–322.

[42] Theodore Romer, Dennis Lee, Brian Bershad, and Bradley Chen. 1994. Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware. In *OSDI*. 255–266.

[43] Shai Rubin, David Bernstein, and Michael Rodeh. 1999. Virtual Cache Line: A New Technique to Improve Cache Exploitation for Recursive Data Structures. In *CC*, Vol. 1575. 259–273.

[44] Sriram Sankaranarayanan. 2020. Reachability Analysis Using Message Passing over Tree Decompositions. In *CAV*. 604–628.

[45] Timothy Sherwood, Brad Calder, and Joel Emer. 1999. Reducing cache misses using hardware and software page placement. In *ICS*. 155–164.

[46] Khalid Thabit. 1982. *Cache management by the compiler*. Rice University.

[47] Mikkel Thorup. 1998. All Structured Programs have Small Tree-Width and Good Register Allocation. *Inf. Comput.* 142, 2 (1998), 159–181.

[48] Thomas van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob. 2006. *Computing treewidth with LibTW*. Technical Report.

[49] Raj Vaswani and John Zahorjan. 1991. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *SOSP*. ACM, 26–40.

[50] Chengliang Zhang, Chen Ding, Mitsunori Ogihara, Yutao Zhong, and Youfeng Wu. 2006. A hierarchical model of data locality. In *POPL*. 16–29.

[51] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. 2004. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI*.