

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Principled Design of Indexing Functions for Memory Coloring

Stephan Dübler and Jana Hofmann, *Max Planck Institute for Security and Privacy* (*MPI-SP*); Boris Köpf and Stavros Volos, *Azure Research, Microsoft*

https://www.usenix.org/conference/usenixsecurity26/presentation/duebler

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Principled Design of Indexing Functions for Memory Coloring

Stephan Dübler[1], Jana Hofmann[1], Boris Köpf[2], Stavros Volos[2]

[1]*Max Planck Institute for Security and Privacy (MPI-SP)*
[2]*Azure Research, Microsoft*

Abstract

On shared multi-core CPUs, memory coloring achieves microarchitectural isolation by partitioning resources between trust domains. Previous work has shown that memory coloring can, in principle, be used to isolate multiple components simultaneously, e.g., both the last-level (L3) cache and DRAM. The number of colors obtained by these methods depends on the algebraic properties of the indexing functions, though, and today's off-the-shelf CPU designs are often not suited for multi-component memory coloring.

We develop algorithms to automatically synthesize indexing functions that guarantee a minimum number of colors. Given a set of typical workloads and some context on the CPU design, our algorithms compute a new linear indexing function that supports the requested number of colors while maximizing the performance on the workloads. Our approach is based on the observation that the number of colors depends on the overlap of the algebraic *kernels* of the involved indexing functions. Building on this observation, we translate the requirements that the CPU design imposes on the function into algebraic constraints that our algorithms enforce.

In a case study on a 16-core server-class CPU, we show that our framework yields a coloring scheme that partitions the L3 cache and DRAM banks, increasing the number of colors from 1 to 16 while incurring less than 2.5% performance overhead, on average across SPEC and PARSEC benchmarks.

1 Introduction

Microarchitectural side channels continue to pose a threat, with a steady stream of new vulnerabilities and retrospectively applied mitigations. The primary targets for microarchitectural side-channel attacks are shared components of the memory hierarchy, for example caches. In modern cloud environments, physical cores are private to each tenant, which prevents attacks via core-local components like L1 caches. Significant parts of the microarchitecture (e.g., last-level (L3) caches, coherence directories, and DRAM) are still shared.

For those shared components, spatial partitioning protects against side channels by assigning each tenant their own portion of the component. Spatial partitioning is achieved either by hardware-based techniques (currently limited to L3 caches [33]) or memory coloring, a software-based technique.

Memory Coloring. Memory coloring assigns colors to physical memory such that addresses of different colors are mapped to different cache sets or DRAM banks. By ensuring that different tenants never share a color, one can guarantee isolation between tenants. Memory coloring has been shown to be practical and effective when partitioning components individually [11,34,46] and can even partition *multiple shared components simultaneously* [16,39].

The number of colors achievable with multi-component partitioning is limited by the address bits used in the components' *indexing functions* (which map addresses to resources) [16]. Modern CPU designs often support only one or two colors [16], which means that a CPU could host only two isolated domains. To make memory coloring a realistic option for cloud providers, CPUs need to support significantly more colors, ideally as many as there are cores. With the growing number of cores in server-class CPUs, the number of tenants that are scheduled on the same machine will only increase.

Indexing Functions for Memory Coloring. In this work, we propose to design indexing functions specifically for multi-component coloring. Traditionally, the primary focus of indexing function design lies on performance, which aims to uniformly distribute memory accesses across the component's resources. Instead, we shift the focus and investigate how we can design indexing functions that first of all guarantee a given minimum number of colors and as a secondary requirement optimize performance.

Problem Statement. The problem of designing indexing functions for coloring has not been studied systematically so far. Previous work suggested small, manual modifications

to existing indexing functions to increase the number of colors [16]. This neglects the performance implications of these modifications. It also does not scale to complex CPU designs, which impose multiple constraints on the coloring scheme and, in extension, also on the indexing functions.

Modern architectures consist of multiple hierarchical components, each with their own indexing function. An L3 cache, for instance, is often organized in slices, which are also indexed by a function. Furthermore, each core has core-local (i.e., private) components like L1 and L2 caches, which should not be inadvertently partitioned by a coloring scheme. Finally, the coloring scheme is constrained by additional architectural demands, e.g., that pages should not be split across colors.

If we want to increase the number of colors by replacing one of the indexing functions involved, we need to consider the system context and how it influences the coloring scheme. The complexity of the surrounding system suggests solving this task algorithmically rather than with manual tweaks.

The question is thus: *Can we automatically synthesize an indexing function that guarantees a minimum number of colors, adheres to system constraints, and does not compromise on performance?*

Approach. We address this question by proposing algorithms that automatically synthesize coloring-friendly and performant indexing functions.

We first focus on the performance aspect by synthesizing indexing functions that evenly distribute memory accesses across the component's resources (e.g., the cache sets of the cache). We formulate this as an optimization problem on the access traces produced by a given set of workloads and solve it using an approximation algorithm.

In the next step, we modify the algorithm to accommodate additional coloring requirements. We describe the coloring requirements algebraically and adapt the algorithm such that they are satisfied by design, provided they are mathematically satisfiable. This is combined with a best-effort approach to ensure performance is not sacrificed in the process.

Lastly, we conduct a case study, in which we synthesize several alternative L3 indexing functions in the context of other, previously reverse-engineered Intel indexing functions. Our results illustrate how the achievable number of colors and performance vary depending on the complexity of the constraints imposed by the surrounding system. We furthermore show experimentally that the coloring schemes derived from the synthesized functions indeed simultaneously partitions all microarchitectural components considered as context.

Contributions. We make the following contributions:

- We provide an algorithm that, given a set of workloads, synthesizes an indexing function that uniformly distributes the workloads' accesses across the component's resources.
- We develop algebraic conditions that guarantee that the synthesized function supports a minimum number of colors and adapt the baseline algorithm accordingly.
- We demonstrate the efficacy of our framework through a case study where we synthesize L3 cache indexing functions for several CPU configurations, including a 16-core server-class CPU. Our performance and security analysis shows that it is possible to derive coloring schemes that jointly partition the L3 cache and DRAM banks while providing a sufficient number of colors, at small performance overhead.

2 Memory Coloring and Indexing Functions

We use this section to give an overview of memory coloring and introduce the formal constraints on a coloring scheme (following the definitions of [16]). We then discuss how the coloring scheme depends directly on the indexing functions of the involved components and deduce coloring requirements for the design of new indexing functions.

2.1 Memory Coloring and Indexing Functions

Threat Model. In public clouds, several tenants (or, more generally, trust domains) run simultaneously on a server. Each trust domain is typically assigned its own set of CPU cores, but off-core components like the last-level (L3) cache, coherence directories, and DRAM are shared. The hypervisor is not adversarial. We consider an attacker who controls one of the trust domains and monitors the usage of these shared components with eviction-based side channels. We do not consider attacks based on overall congestion (e.g., utilization-based) on coarse-grained shared resources (e.g., cache ports and banks, on-chip interconnects, and the memory bus) as well as attacks via networking devices, shared storage, physical (e.g., power) side channels.

Resource Partitioning. Memory coloring is a software-based technique that spatially partitions the resources of shared components. When partitioning caches, for example, each cache set should be used exclusively by one trust domain. This is achieved by assigning colors to physical addresses. A coloring scheme achieves isolation if no two addresses of different colors are mapped to the same resource by the component's indexing function. Finally, the memory manager of the hypervisor or host OS assigns each memory color to a trust domain. The number of colors thus determines how many trust domains can compute simultaneously and in an isolated fashion on the same CPU.

Challenges of Memory Coloring. The challenge of memory coloring lies in the fact that multiple shared components need to be partitioned simultaneously – not only the L3 cache, but also DRAM and cache coherence directories. Additionally, the coloring scheme should not inadvertently partition core-local (i.e., private) components like L1 or L2 caches,

which would lead to performance loss due to underutilization of these components. Finally, we need to adhere to architectural constraints, e.g., that pages cannot be split across colors.

Modeling Memory Coloring. Formally, we model the memory as the vector space $\mathbb{F}_2^n$ containing all n-bit addresses. In this setting, a coloring scheme with 2^m colors, assuming $m \leq n$, can be expressed as a function $h : \mathbb{F}_2^n \to \mathbb{F}_2^m$, i.e., two addresses x_0 and x_1 have the same color if $h(x_0) = h(x_1)$. The preimages $P_h := \{h^{-1}(r) \mid r \in \mathbb{F}_2^m\}$ of h form a partition of the memory space: each class $C_r \in P_h$ contains all addresses assigned the same color r. Importantly, different functions can define the same partition. For example, the function $h(x) = x^0$, which projects the address to its least significant bit, defines the same partition as $h'(x) = \neg x^0$, which projects on and then negates the least significant bit.

Indexing Functions. Indexing functions define how physical addresses are mapped to the resources of a component. For example, a typical L3 set indexing function projects on the 11 least significant bits (ignoring the block-offset bits):

$$f_{L3}(x) = x^6,\, x^7,\, \ldots,\, x^{16}$$

Above, each comma-separated term defines an *output bit* of f_{L3}. On a typical machine with 48 usable address bits, f_{L3} is a function of type $\mathbb{F}_2^{48} \to \mathbb{F}_2^{11}$, indexing 2048 cache sets. Most indexing functions are linear functions, i.e., they either project on single bits or XOR ($\oplus$) multiple bits, but they do not use AND or OR operations. Linear functions can be represented using matrices, where each row defines an output bit. A column contains 1 if the address bit is present in the corresponding output bit.

Example 1. *The following XOR notation and matrix notation represent the same linear function.*

$$f : \mathbb{F}_2^3 \to \mathbb{F}_2^2 \qquad A_f = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
$$f(x) = x^1 \oplus x^2,\, x^0$$

As for coloring functions, the preimages P_f of an indexing function define a partition of the memory, where each class contains all addresses mapped to the same resource. A special class in P_f is the *kernel* $ker(f)$, which is the class $f^{-1}(0)$ of addresses mapped to 0. The kernel uniquely characterizes P_f, which means that two functions with the same kernel define the same partition. From $ker(f)$, we can directly construct a function f' with $ker(f') = ker(f)$: if K is the matrix whose columns form a basis of $ker(f)$, the matrix describing f' is $\text{BASIS}(ker(K^T))^T$, where T transposes a matrix (see [16], Sec. 3.2, for an explanation why this is the case).

2.2 Computing Coloring Functions

Isolating Coloring Schemes. A memory coloring function h is *isolating* for a shared component indexed by f if addresses with different colors are mapped to different resources. Formally, we can express isolation in terms of the preimages P_h of the coloring function:

$$\forall C_1, C_2 \in P_h.\, \{f(x) \mid x \in C_1\} \cap \{f(x) \mid x \in C_2\} = \emptyset.$$

The above condition is equivalent to $P_f \sqsubseteq P_h$, which states that every class of P_f is a subset of a class of P_h.

Architectural Constraints. Usually, CPU designs have additional architectural constraints, e.g., that pages cannot be split across colors. For 4KB pages, we define

$$f_{4K}(x) = x^{12},\, x^{13},\, \ldots,\, x^{47},$$

which maps a 48-bit address to the page it is located on. The constraints that pages are not split across colors translates to $P_{f_{4K}} \sqsubseteq P_h$.

Simultaneous Isolation. If we want to simultaneously color multiple components $f_1, f_2, \ldots, f_k$ while satisfying the architectural constraint f_{4K}, we need $P_{f_i} \sqsubseteq P_h$ for all $i \in \{1, 2, \ldots, k, 4K\}$. Additionally, P_h needs to have as many classes as possible to maximize the number of colors. Formally, this is expressed as

$$P_h = P_{f_1} \sqcup P_{f_2} \sqcup \ldots \sqcup P_{f_k} \sqcup P_{f_{4K}}, \tag{1}$$

where $P \sqcup Q$ is the join of two partitions, i.e., the finest partition that is coarser than both P and Q.

Uniform Utilization of Private Components. Besides shared components, a system also contains core-local components like L1 and L2 caches, which are private when cores are not shared. Since core-local components are indexed from memory just like shared ones, a coloring scheme might accidentally partition those as well. This would mean that a trust domain can access only a share of its private cache sets, which would unnecessarily decrease performance. A coloring scheme P_h should thus *uniformly utilize* all private components g (written $P_h \perp P_g$):

$$\forall C_1 \in P_h, C_2 \in P_g.\, \frac{|C_1 \cap C_2|}{|C_1|} = \frac{|C_2|}{|\mathbb{F}_2^n|} \tag{2}$$

The formula requires that the proportion of addresses being mapped to a resource C_2 of g is the same in each color and across the entire memory.

Hierarchical Indexing Functions. Some microarchitectural components are structured hierarchically. The cache sets of L3 caches, for example, are often grouped into several slices, which are indexed by a slicing function. To map an address to its cache set, both functions f_{slice} and f_{L3} are applied to the address, and the output of both together indexes

the cache set. In the matrix representation, this corresponds to stacking the two matrices vertically, which we denote as (f_{slice}, f_{L3}). To achieve isolation of a hierarchical component, we therefore need $P_{(f_{slice},f_{L3})} \sqsubseteq P_h$. The output bits of the slicing function and the set indexing function need to be independent of each other in order to index all cache sets within a slice. This is equivalent to $P_{f_{slice}} \perp P_{f_{L3}}$.

Deriving the Coloring Scheme. In summary, a coloring scheme P_h should be (1) isolating for all shared resources, (2) adhere to architectural constraints, and (3) uniformly utilize all private components. Previous work has shown that for fixed indexing functions, there is a deterministic single solution to these constraints [16]. For off-the-shelf indexing functions (which are not designed with coloring in mind), it is often the case that P_h has very few colors or even just a single one [16]. To make memory coloring applicable and practical, a systematic way to increase the number of colors in a coloring scheme is needed.

2.3 Function Design for Memory Coloring

For this work, we observe that the coloring scheme depends directly on the indexing functions of the shared components, the private components, and the architectural constraints. Thus, by choosing a different indexing function for one of the components, we can potentially increase the number of colors. In the following sections, we investigate how to automatically synthesize a function for a shared component such that the resulting coloring scheme supports a minimum of c colors and still achieves a good hit rate on relevant workloads.

Formally, we assume a given set of functions indexing shared components and architectural constraints $f_1, \ldots, f_l$, a set of functions for private components $g_1, \ldots, g_k$, and potentially a slicing function f_{slice}. We want to synthesize a new indexing function f that satisfies the following requirements.

1. COLORING: The resulting coloring scheme $P_h = P_{(f_{slice},f)} \sqcup P_{f_1} \sqcup \ldots \sqcup P_{f_l}$ has the required number of colors: $|P_h| = c$.
2. UTILIZATION: The resulting coloring scheme P_h uniformly utilizes all private components: $P_h \perp P_{g_i}$ for $1 \leq i \leq k$.
3. HIERARCHICAL INDEPENDENCE: f and f_{slice} can be organized hierarchically, i.e., the slicing function uniformly utilizes f: $P_{f_{slice}} \perp P_f$.
4. PERFORMANCE: f performs well on a given set of workloads.

The first three requirements we have formalized in this section. In the following section, we make the PERFORMANCE requirement more precise before we present our synthesis algorithms.

3 Optimizing for Performance

As a metric to evaluate the performance of an indexing function, we use the *hit rate*, e.g., the cache hit rate or row buffer hit rate. Concretely, we want to maximize the hit rate for a given set of workloads. Previous work observed that in order to maximize the hit rate, we need to uniformly distribute the accesses generated by the workloads across the component's resources [20]. In a cache, for example, we want to evenly spread out accesses across the cache sets. In general, it is not enough to synthesize a function that is *globally* uniform, i.e., one that accesses every resource equally often counted across all workload traces [20]. This would allow mapping a bulk of consecutive accesses to the same resource, which would cause a significant spike in the contention of a single resource and negatively affect the hit rate.

This is why we require *local uniformity*, i.e., that uniformity should also hold in any subsequence of successive accesses. This means that ideally, in any interval whose length equals the number of available resources, every resource should be accessed exactly once.

In this section, we define an optimization problem to synthesize a function that approximates this ideal. Before we define the optimization problem, however, we need to discuss how we represent workload access traces and how we transform them into a probability distribution.

3.1 From Address Traces to Offset Traces

We assume we are given a set of address traces, where each trace represents the accesses to the component generated by one workload. The traces are obtained from workloads that model typical component usage.

3.1.1 Bitflip Patterns

The address traces are highly dependent on the memory allocator: physical addresses are assigned in pages, but the pages may be scattered across the memory. The memory allocation will differ depending on the machine, the operating system, and other parties computing on the machine. Thus, two traces produced by the same binary might contain entirely different addresses, depending on which pages were assigned to the process. A function uniformly distributing the addresses of one trace might perform badly for another trace. We therefore abstract away concrete addresses and focus on *offsets* between addresses. Given an address trace $t = a_0, a_1, a_2, a_3, \ldots$, the offset trace $o_0, o_1, \ldots$ is defined as $o_0 = a_0 \oplus a_1$, $o_1 = a_1 \oplus a_2, \ldots$. The offset trace describes the bitflips that occur between pairs of consecutive addresses.

An advantage of offset traces is that they better capture repeating patterns within a trace. Many workloads contain linear traversals of arrays, for instance. These so-called strides will appear repeatedly, starting from different addresses. Off-

sets abstract away from the location where an array is stored and produce similar bitflip patterns across all strides. This is illustrated by the following example.

Example 2. *Consider the following two snippets of different address traces accessing consecutive memory locations:*

$$t_1 = \texttt{110000}, \texttt{110001}, \texttt{110010}, \texttt{110011}, \texttt{110100}$$
$$t_2 = \texttt{001011}, \texttt{001100}, \texttt{001101}, \texttt{001110}, \texttt{001111}$$

Translated to offset traces, we obtain the following two traces:

$$t_1' = \texttt{000001}, \texttt{000011}, \texttt{000001}, \texttt{000111}$$
$$t_2' = \texttt{000111}, \texttt{000001}, \texttt{000011}, \texttt{000001}$$

The two offset traces contain the same offsets, just in a different order: in a linear stride, `000001` *is the most likely offset, followed by* `000011` *and* `000111`.

Considering Example 2, we expect that different address traces have offset traces with a similar distribution of offsets.

3.1.2 Uniformity in Terms of Offsets

We have established that offset traces better surface repeating bitflip patterns, but in the end, the function should perform well on the address traces, not the offset traces. However, since we work with *linear* functions, we can use the offset trace to infer the performance of a function on the address trace. Assume we want to synthesize a function mapping to 2^m resources. As discussed at the beginning of the section, ideally, we would like to achieve local uniformity, i.e., that in any 2^m-long subsequence of distinct addresses, no two addresses are mapped to the same resource. We can formulate this condition equivalently on offsets. Let $a_i, \ldots, a_{i+2^m-1}$ be a sequence of pairwise different addresses. We can express any a_{i+j} in terms of its sum of offsets from a_i:

$$a_{i+j} = a_i \oplus o_i \oplus \ldots \oplus o_{i+j-1}$$

Let o^i_j be the sum $o_i \oplus \ldots \oplus o_{i+j-1}$. We define o^i_0 as the zero vector 0. There is a collision in $a_i, \ldots, a_{i+2^m-1}$ if there are $j,k \in [0, \ldots, 2^m-1]$ such that

$$\begin{aligned} f(a_{i+j}) = f(a_{i+k}) &\equiv f(a_i \oplus o^i_j) = f(a_i \oplus o^i_k) \\ &\equiv f(a_i) \oplus f(o^i_j) = f(a_i) \oplus f(o^i_k) \\ &\equiv f(o^i_j) = f(o^i_k) \end{aligned}$$

The above transformation holds because of the linearity of f, which gives us $f(x \oplus y) = f(x) \oplus f(y)$.

From the above equations, we get that a 2^m-long interval starting at index i does not contain a collision if all offset sums $o^i_0, o^i_1, \ldots, o^i_{2^m-1}$ are mapped to different resources by f. Since $o^i_0 = 0$ and for linear functions, $f(0) = 0$, we obtain that *all* o^i_j, o^i_k *with* $j \neq k$ *and* $j,k > 0$ *should be mapped to different resources, and none of them to* 0.

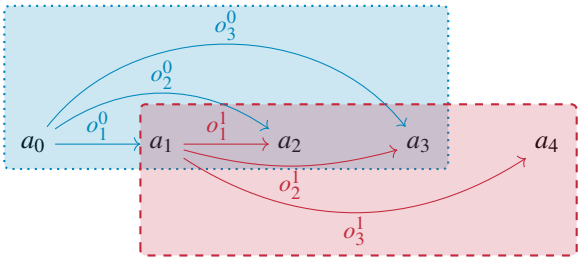


Figure 1: Sliding windows for $2^m = 4$. The first (blue, dotted) window produces 1, 2, and 3-step offsets starting from address a_0, the second (red, dashed) produces 1, 2, and 3-step offset sums starting from address a_1.

This condition on offsets ensures local uniformity, and we want to synthesize a function that comes as close as possible to satisfying this property. We therefore formulate this goal as a probabilistic optimization problem, for which we first transform the offset traces into a probability distribution.

3.2 Defining the Optimization Problem

3.2.1 Probability Model

We use a discrete probability space, in which each offset is assigned a probability directly, independent of the history. The set of all offsets is the same as the address space. Therefore, a probability distribution on offsets is a function $P : \mathbb{F}_2^n \to [0,1]$. We decided to use discrete, direct probabilities to keep the probabilistic aspect of our algorithm simple and general.

A simple way of transforming offset traces into a discrete probability distribution would be to count the number of occurrences of each offset and compute its share of the total number of offsets. However, the offset-based uniformity criterion developed in Section 3.1.2 argues about *sums* o^i_j of offsets, not just single-step offsets. If we just counted the number of occurrences of single-step offsets, a frequent offset sum that never occurs as a single-step offset would be assigned probability 0. Therefore, we instead count all offset sums obtained by adding at most $2^m - 1$ many consecutive single-step offsets. We do so by traversing every address trace using sliding windows of length 2^m, as illustrated in Figure 1. Each window produces $2^m - 1$ many offsets, and each trace t contains $l = |t| - (2^m - 1)$ many windows. We define $P(o)$ as the number of times o was observed across all windows divided by $l \cdot (2^m - 1)$. Elements from $\mathbb{F}_2^n$ that have not been observed in any of the sliding windows are assigned a probability of 0. From now on, we call any $o \in \mathbb{F}_2^n$ an offset, independently of it being a single-step offset or an offset sum.

3.2.2 Uniformity Probabilistically

Assume we picked a function f that distributes all offsets to the 2^m resources. We define the probability mass M_r of

a resource r as the sum of the probabilities of all offsets mapping to r:

$$M_r = \sum_{o \in f^{-1}(r)} P(o)$$

We randomly draw a sequence of offsets $o^i_1, \ldots, o^i_{2^m-1}$. According to the formulation of uniformity in terms of offsets (as described in Section 3.1.2), all $o^i_1, \ldots, o^i_{2^m-1}$ should be mapped to different resources and none of them to 0. One way to satisfy this requirement is $f(o^i_1) = 1, f(o^i_2) = 2, \ldots, f(o^i_{2^m-1}) = 2^m - 1$ (each $1, 2, \ldots, 2^m - 1$ refers to a different resource). The probability of this event occurring is $M_1 \cdot M_2 \cdot \ldots \cdot M_{2^m-1}$. Any permutation of $M_1, \ldots, M_{2^m-1}$ is fine as well, so the probability of not observing a collision is the sum of the $(2^m - 1)!$ many permutations of the sequence:

$$(2^m - 1)! \cdot M_1 \cdot \ldots \cdot M_{2^m-1}$$

The probability masses M_i are dependent on the function we choose, so the optimization task is the following:

Optimization Goal:
Find a linear function $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ that maximizes the probability $(2^m - 1)! \cdot M_1 \cdot \ldots \cdot M_{2^m-1}$.

3.2.3 Example

We present a small example that illustrates how synthesizing a function from offset probabilities according to the above optimization goal ensures a uniform distribution of addresses to resources.

Assume we are given an address trace with just 4 different addresses: `11`, `10`, `01`, `00`. Assume 50% of the addresses in the trace are `11`, 30% are `10`, 15% are `01`, and 5% are `00`. Intuitively, to achieve uniformity, `11` and `00` should be mapped to the same resource, as well as `10` and `01`. Because of the distribution, the offsets we will see the most are:

1. `00` = `11` $\oplus$ `11` 2. `01` = `11` $\oplus$ `10` 3. `10` = `11` $\oplus$ `01`

Assume we want to synthesize a linear function $f : \mathbb{F}_2^2 \to \mathbb{F}_2^1$, i.e., we have two resources 0 and 1. The optimization goal tells us to choose a function f such that M_1 is as large as possible. $f($`00`$)$ has to be assigned to resource 0 because of the linearity. We therefore assign the offset with the second-highest probability to resource 1, which is `01`. We also add `10` to resource 1, leaving `11` to go to resource 0 again (because of the linearity). Like that, we have synthesized a linear function that maps `11` and `00` to the same class, which are exactly the addresses with the highest probability and the lowest one.

3.2.4 Solving the Optimization Problem

Ignoring the additional constraint that the mapping from offsets to resources needs to be produced by a linear function, the type of optimization problem derived in this section is well established and can be solved with the Lagrange optimization method [35]. Using the Lagrange method, we obtain the result that we should pick a function that a) minimizes M_0 and b) minimizes the standard deviation between all other M_i, i.e., getting as close to $M_1 = \ldots = M_{2^m-1}$ as possible. The detailed steps applying the Lagrange method can be found in the appendix in Section A.

Our optimization problem is very close to the multi-ary variant of the well-known number partitioning problem [18, 31]. The number partitioning problem asks to distribute a set of n values across m sets such that the standard deviation between the sums in each of the sets is minimized. The number partitioning problem has been dubbed the “easiest NP-hard problem" because of the various efficient approximation algorithms [14, 22, 24]. In our case though, we have an additional requirement that the distribution of offsets to resources needs to be computable by a linear function. As we will see in the next section, this requirement adds significant additional constraints.

4 Greedy Synthesis of Linear Functions

We approximate the optimization problem described in Section 3.2 using a greedy algorithm inspired by algorithms proposed for the number partitioning problem [14]. This algorithm serves as a baseline that we will then extend to also guarantee the required number of colors. Instead of defining a function f right away, our algorithm constructs its partition $P_f = \{C_0, \ldots, C_{2^m-1}\}$, i.e., it groups all offsets of the input space $\mathbb{F}_2^n$ into 2^m many resource classes. Here we leverage the fact that in the case of linear functions, it is sufficient to know P_f to compute f (as described in Section 2.1).

Overview of the Algorithm. Intuitively, our algorithm is quite straightforward. Given a probability distribution P over offsets, iteratively assign all offsets to the classes $C_0, \ldots, C_{2^m-1}$, keeping the probability mass of the class representing the kernel as small as possible and the masses of all other classes as uniform as possible (as described in Section 3.2). We proceed in two steps: first, we pick the 2^m most likely offsets that we want to assign to different classes. Then, we repeatedly pick from the remaining offsets the one with the highest probability and assign it to the class with the lowest probability mass (other than the kernel).

Linearity Constraint. The algorithm is complicated by the fact that we need the resulting partition to be computable by a linear function, which poses the additional constraint that

$$\forall a, b : f(a \oplus b) = f(a) \oplus f(b).$$

With this constraint, if we assign some offset o_1 to the class C_1 and o_2 to C_2, we know to which class we need to assign offset $o_1 \oplus o_2$, namely to the class representing $f(o_1) \oplus f(o_2)$. In

particular, if o_1 is in the kernel class, then o_2 and $o_1 \oplus o_2$ need to be in the same class. Furthermore, the constraint implies that all classes in the partition must be of equal size. To adhere to the linearity constraint, we ensure that every intermediate partial assignment of offsets to classes satisfies the constraint. Thus, with every new offset assigned to a class, we compute the additional offset assignments implied by the constraint.

4.1 Step 1: Pick Class Representatives

Let $\mathbb{D}$ be the list of offsets ordered by their probability. Initially, all classes $C_0, \ldots, C_{2^m-1}$ are empty. In the first round, depicted in Algorithm 1, we pick one offset per class. To class C_0 we assign 0, which will thus be the kernel class. The offsets we pick for the other classes are the ones with the highest probability. These are the offsets that occur particularly often and should therefore be assigned to different classes to avoid collisions. We call these initial offsets *class representatives*.

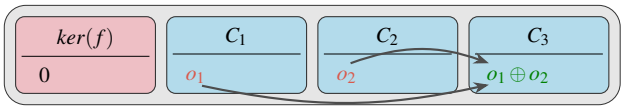
Algorithm 1 Picking Class Representatives

Input : Offsets sorted by probability $\mathbb{D}$
Output: Partial partition P_f with one offset per class
$P_f \leftarrow [\{0\}]$ ▷ *Initializing the kernel*
while $|P_f| < 2^m$ **do**
  $c \leftarrow \mathbb{D}.pop()$ ▷ *Offset with highest probability*
  $P_f.append(\{c\})$
  for $\{c'\} \in P_f$ **do**
    $c'' \leftarrow c \oplus c'$ ▷ *Linear combination*
    $P_f.append(\{c''\})$
    $\mathbb{D} \leftarrow remove(c'', \mathbb{D})$
return: P_f

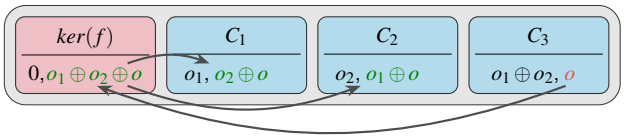
Applying the Linearity Constraint. Because of the linearity constraint, we cannot freely pick all class representatives. Assume we assign o_1 to C_1 and o_2 to C_2. Then $o_3 = o_1 \oplus o_2$ must be assigned to a third class C_3. This is because $f(o_1) \neq f(o_2)$ and both $f(o_1)$ and $f(o_2)$ are unequal to 0, so $f(o_1) \oplus f(o_2)$ is unequal to $f(o_1), f(o_2)$, and $f(0)$. This is depicted in Figure 2a. Generally speaking, whenever we choose a new class representative o, all linear combinations of o with previously chosen class representatives need to be assigned to their own class. As a result, the space of class representatives forms a proper subspace of $\mathbb{F}_2^n$. Lastly, we initialize the probability mass M_i of each C_i with the probability $P(o_i)$ of the offset picked as class representative.

4.2 Step 2: Distribute Remaining Offsets

In the next step, depicted in Algorithm 2, we repeatedly pick the offset with the highest probability and assign it to the class with the lowest probability mass.



(a) Linearity implications when picking class representatives. The green offset $o_1 \oplus o_2$ needs to be added to a class different from C_1 and C_2.



(b) Linearity implications when distributing remaining offsets. Adding new offset o_3 to class C_3 constructs a new kernel offset $(o_1 \oplus o_2 \oplus o_3)$, from which we get new class offsets $(o_2 \oplus o_3$ and $o_1 \oplus o_3)$.

Figure 2: Adhering to Linearity Constraints.

Algorithm 2 Synthesizing a Linear Function $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$

Input : Sorted probability distribution of offsets $\mathbb{D}$, P_f from Algorithm 1
Output: Full partition P_f
while $\mathbb{D} \neq \emptyset$ **do**
  $c \leftarrow \mathbb{D}.pop()$
  $C \leftarrow lowestProbability(P_f.remove(ker(f)))$
  $C.add(c)$ ▷ *Class with lowest probability*
  $P_f \leftarrow satisfyLinearityInvariant(P_f)$
if P_f is not a complete partition of $\mathbb{F}_2^n$ **then**
  fill $ker(f)$ under linearity constraint
return: P_f

Linearity Invariant. To adhere to the linearity constraint, we maintain the invariant that the space of all assigned offsets is always a proper subspace of $\mathbb{F}_2^n$, i.e., if offsets o_1 and o_2 were assigned to classes already, then so was $o_1 \oplus o_2$. Initially, after picking all class representatives, the invariant is satisfied.

Now, assume we want to add an offset o to class C_3. To again satisfy the invariant, we need to compute the linear implications entailed by this operation. We do so in two steps, illustrated in Figure 2b.

1. Let o_3 be the representative offset of C_3. Because of the linearity, we need to add $o \oplus o_3$ to the kernel: $f(o) = f(o_3)$ implies $f(o \oplus o_3) = 0$.

2. Next, for each class C_i, if $o_i \in C_i$, then add $o_i \oplus o \oplus o_3$ to C_i. This operation re-establishes the invariant that the space of all assigned offsets forms a subspace of $\mathbb{F}_2^n$.

With this approach, we also make sure that all classes are always of the same size. Finally, whenever we add an offset o to a class C_i, we add the probability $P(o)$ to the probability mass M_i and remove o from $\mathbb{D}$ for proper bookkeeping.

Finalizing the Function. If we want to synthesize a function $\mathbb{F}_2^n \to \mathbb{F}_2^m$, each class needs to contain $n-m$ linearly

independent offsets. If the required dimension is reached, the algorithm terminates and uses the kernel to compute a function that implements the final partition. However, before this is the case, the algorithm usually reaches a point at which all remaining offsets have a probability of 0 (because even when counting offsets in windows, we only observe a subset of all possible offsets). At this point, we fill the kernel with linearly independent vectors to achieve the required dimension.

5 Synthesis for Memory Coloring

We now turn to the COLORING requirement, i.e., how we can synthesize a function that guarantees that the coloring scheme has at least c colors. According to Equation (1) in Section 2.2, let $f_1, \dots f_l$ be a collection of functions containing both architectural constraints and the indexing functions of other shared components. We aim to synthesize a function f such that the join $P_h = P_f \sqcup P_{f_1} \dots \sqcup P_{f_l}$ has at least 2^c many classes. Since the join operation is associative, we first compute $P_{f'} = P_{f_1} \dots \sqcup P_{f_l}$ and simplify the problem to finding f such that $P_f \sqcup P_{f'}$ has at least 2^c many classes.

Our algorithm is based on the algebraic observation that the size of $P_f \sqcup P_{f'}$ is determined by the overlap of the vector spaces spanned by the kernels of f and f'. We first expand on this observation (Section 5.1) and subsequently describe how we adapt the algorithm accordingly (Section 5.2).

5.1 Coloring Requirement Algebraically

The kernel $ker(f)$ of a linear function always forms a proper subspace of the function's domain. The dimension $dim(ker(f))$ of this space is defined by the number of linearly independent vectors in $ker(f)$. Using a few simple algebraic transformations (given in detail in Section B), we obtain the following correlation between the number of colors and the dimension of the intersection of f's and f''s kernels.

Proposition 1. *Given $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ and $f' : \mathbb{F}_2^n \to \mathbb{F}_2^{m'}$, $P_f \sqcup P_{f'}$ has 2^c many classes if*

$$dim\left(ker(f) \cap ker(f')\right) = n - m' - m + c$$

With the above , the intersection of the kernels of f and g must have a dimension of at least $k = n - m' - m + c$. This also gives us an upper bound on the number of colors we can enforce, namely $min(m, m')$.

Example 3. *Consider the following two functions.*

$$f(x) = x^0,\ x^1 \oplus x^2,\ x^4$$
$$f'(x) = x^0,\ x^1 \oplus x^2 \oplus x^3,\ x^3$$

For this example, assume f and f' are defined over input domain $\mathbb{F}_2^5$. The kernels of the functions are the following:

$$ker(f) = \{\texttt{01000},\ \texttt{00110},\ \texttt{01110},\ \texttt{00000}\}$$
$$ker(f') = \{\texttt{10000},\ \texttt{00110},\ \texttt{10110},\ \texttt{00000}\}$$

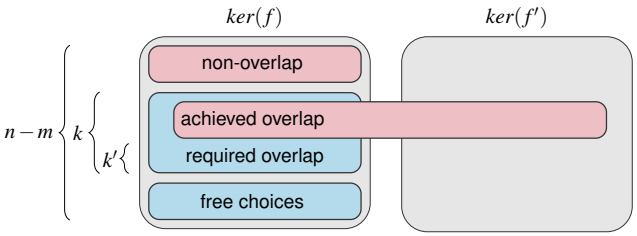


Figure 3: Intermediate state of constructing $ker(f)$. $ker(f)$ overlaps with $ker(f')$ in $k - k'$ dimensions, and we have already picked some kernel vectors that do not overlap with $ker(f')$. In total, the overlap needs to have a dimension of k.

The fact that `01000` is in the kernel of f tells us that for a given address x, if we flip bit x^3, the resulting address is mapped to the same class as x. The intersection of the two kernels is $\{\texttt{00000}, \texttt{00110}\}$, which has a dimension of 1. Thus, f and f' together support $2^2 = 4$ colors.

5.2 Adapting the Algorithm

Given function $f' : \mathbb{F}_2^n \to \mathbb{F}_2^{m'}$ and a number k informing us how large the overlap of kernels should be, we now adapt the algorithm from Section 4 such that it enforces the required overlap when constructing $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$. We assume that $k \leq min(m, m')$, i.e., that it is theoretically possible to enforce the required number of colors. The algorithm proceeds as before, but when distributing the offsets, the last k dimensions of the kernel are filled with k independent offset vectors from $ker(f')$. We proceed with the performance-oriented algorithm for as long as possible, such that enforcing the coloring requirement has a minimal impact on the performance. The algorithmic challenge lies in determining the point at which we have to add kernel offset vectors from $ker(f')$ to $ker(f)$.

Step 1: Pick Class Representatives. Remember that the first step of the algorithm picks 2^m class representatives. The class representatives form a proper subspace of $\mathbb{F}_2^n$. With every representative we pick, we add a new dimension to that subspace. Importantly, any offset that is in the space of the class representatives cannot be added to the kernel of f anymore. Thus, we maintain a set of basis vectors B_{rep} of the class representatives and a set of basis vectors $B_{ker(f')}$ of the kernel of f'. If at some point, more than $|B_{ker(f')}| - k$ vectors of $B_{ker(f')}$ lie in the span of B_{rep} (i.e., are linearly dependent on the vectors in B_{rep}), we know that there are not enough vectors left in $ker(f')$ that could be added to $ker(f)$. Then, we backtrack the last class representative we picked and pick the one with the next highest probability.

Step 2: Distribute Remaining Offsets. For the second step, we proceed similarly to before, i.e, we monitor the distribution of offsets until we need to step in to guarantee the required

number of colors. With each offset we add to a specific class, we add a new dimension to $ker(f)$. After each step, we check how much $ker(f)$ and $ker(f')$ overlap already, which tells us how many more dimensions of overlap we need to achieve. Let's say this number is k' with $k' \leq k$ (illustrated in Figure 3). Since we know that the final dimension of $ker(f)$ will be $n-m$, we stop at the point where $k' = (n-m) - dim(ker(f))$. At this point, we determine k' basis vectors of $ker(f')$ that are independent of $ker(f)$ and add them to $ker(f)$. Then, we compute f from the final $ker(f)$ as before.

5.3 Hierarchical Functions

We now turn to the case where the function f we synthesize is embedded hierarchically in a slicing structure. In a sliced component, each resource is indexed by the combined function (f_{slice}, f) consisting of the slicing function f_{slice} and the second-level indexing function f. As discussed in Section 2.2, to achieve simultaneous isolation of the hierarchical and another shared component f', we need to compute $P_h = (f_{slice}, f) \sqcup f'$, for which the coloring algorithm needs to be slightly adapted.

Adapting the Algorithm. In the presence of a slicing function, not the kernel of f has to overlap in the required dimension, but the kernel of (f_{slice}, f). The kernel of (f_{slice}, f) is given by $ker(f_{slice}) \cap ker(f)$. Thus, every kernel vector we add from $B_{ker(f')}$ to $ker(f)$ needs to be in $ker(f_{slice})$ as well. This can be achieved by initially restricting $B_{ker(f')}$ to $B_{ker(f')} \cap ker(f_{slice})$. Furthermore, (f_{slice}, f) defines more output bits than f alone, therefore, according to Proposition 1, the needed overlap of $ker(f')$ and $ker((f_{slice}, f))$ reduces by m'' many dimensions if $f_{slice} : \mathbb{F}_2^n \to \mathbb{F}_2^{m''}$.

Filtering of Traces. When f is the second-level indexing function, the addresses reaching f are already filtered by slice. This needs to be considered when synthesizing f. For every workload, we split its trace into $2^{m''}$ traces, one for each of the resources indexed by f_{slice}. As a result, we only compute offsets between addresses that are mapped to the same slice.

6 Hierarchical Independence and Uniform Utilization

As a last step, we discuss how to satisfy the HIERARCHICAL INDEPENDENCE and UNIFORM UTILIZATION requirements when synthesizing f.

6.1 Hierarchical Independence

The HIERARCHICAL INDEPENDENCE requirement states that the indexing functions of a hierarchical component must be linearly independent. As discussed in Section 2.2, independence of f and a slicing function f_{slice} is equivalent to $P_{f_{slice}} \perp P_f$. We show how do adapt the baseline synthesis algorithm from Section 4 to guarantee that the synthesized function f satisfies $P_{f_{slice}} \perp P_f$. Initially, we ignore the modifications needed to satisfy the COLORING requirement, but we will later argue that the two requirements can be enforced simultaneously. As in Section 5, we formulate the requirement algebraically and adapt our algorithm to enforce it.

6.1.1 Algebraic Formulation

Uniform utilization $P_{f_{slice}} \perp P_f$ is equivalent to requiring that the rowspaces of the matrices of f_{slice} and f do not overlap [16]. Intuitively, an output bit of one function should not be obtainable by combining output bits of the other.

Example 4. *Consider the following two functions f and f_{slice}.*

$$f(x) = x^0,\ x^1,\ x^2$$
$$f_{slice}(x) = x^0 \oplus x^1,\ x^2 \oplus x^3$$

f and f_{slice} are not uniformly utilizing as the output bit $x^0 \oplus x^1$ of f_{slice} is a linear combination of f's output bits x^0 and x^1.

Using simple algebraic transformations (detailed in Section C), we arrive at the following proposition.

Proposition 2. *Given $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ and $f_{slice} : \mathbb{F}_2^n \to \mathbb{F}_2^{m'}$ with $rowspace(f) \cap ker(f) = 0$, we have $P_{f_{slice}} \perp P_f$ if and only if*

$$rowspace(f_{slice}) \subseteq ker(f).$$

This proposition gives us a recipe to adapt the synthesis algorithm: the kernel of f needs to contain $rowspace(f_{slice})$ and should not overlap with $rowspace(f)$.

6.1.2 Adapting the Algorithm

We again modify the two phases of the algorithm, picking class representatives and distributing offsets across classes. Let B_{rowsfs} be a set of basis vectors of $rowspace(f_{slice})$.

Step 1: Pick Class Representatives. The vector space of class representatives needs to be strictly disjoint from the kernel of f (except for the 0 offset). Thus, whenever we add a new class representative, we check if the offset is in the span of the offsets of B_{rowsfs}. If so, then selecting this offset as a class representative would mean we could no longer fully add B_{rowsfs} to the kernel of f. Thus, we discard that offset and pick the one with the second-highest probability.

Step 2: Distribute Remaining Offsets. Before we distribute the remaining offsets according to their probabilities, we add all offsets from B_{rowsfs} to the kernel of f. Then, we compute the linear implications of adding these offsets to

$ker(f)$ (which adds $|B_{rowsfs}|$ many offsets to each class) and update the probability masses of the classes accordingly. Afterwards, we proceed as before and distribute the remaining offsets according to their probabilities.

Combination with the COLORING Requirement. The modifications for both the COLORING and the HIERARCHICAL INDEPENDENCE requirement require a certain set of offsets to be in the kernel of f. It is thus easy to combine both requirements algorithmically. However, the more offsets need to be in the kernel of f, the fewer kernel offsets we are free to choose according to their probability. Satisfying more requirements will thus decrease the performance of f.

Post-Hoc Analysis. As $\mathbb{F}_2$ is a degenerate space, it might happen that $rowspace(f) \cap ker(f) \neq 0$ does not hold after this procedure, meaning that Proposition 2 could not be applied. We then adapt $ker(f)$ to regain the property $P_{f_{slice}} \perp P_f$: for any vector $v \in rowspace(f_{slice}) \cap rowspace(f) \cap ker(f)$, we construct a vector v' with $v \oplus v' \neq 0$ and replace v with v' in $ker(f)$. This removes v from $rowspace(f)$ since $rowspace(f) = ker(f)^{\perp} = \{w \mid \forall w' \in ker(f) : w \oplus w' = 0\}$ and hence ensures $P_{f_{slice}} \perp P_f$.

6.1.3 Non-linear Slicing Functions

So far, we assumed all functions to be linear. Modern server-class CPUs often have a non-power-of-two number of slices, which need to be indexed by non-linear functions. Luckily, these functions often have the following structure [28]:

$$f_{slice}(x) = f^l_1(x), \ldots, f^l_{m_1}(x), \\ f^{nl}(\, f^l_{m_1+1}(x), \ldots, f^l_{m_1+m_2}(x)\,)$$

The first m_1 many output bits are defined by linear functions. The remaining non-linear output bits are obtained by first feeding the address into m_2 many linear functions, whose results are then fed into a non-linear mixer f^{nl} that produces m_3 many output bits. The final function thus defines $m = m_1 + m_3$ many bits. For functions of this structure, we obtain the following result straightforwardly:

Proposition 3. *If* $P_{(f^l_1,\ldots,f^l_{m_1+m_2})} \perp P_f$ *then* $P_{(f^l_1,\ldots,f^l_{m_1},f^{nl})} \perp P_f$.

The proposition states that in order to ensure that f can be used hierarchically together with f_{slice}, it is enough to ensure that it can be used together with all the linear components of f_{slice}. Intuitively, this proposition holds because the non-linear mixer f^{nl} merges the classes defined by $(f^l_{m_1+1}, \ldots, f^l_{m_1+m_2})$: by definition, for any two addresses x and y, if $f^l_{m_1+1}(x) = f^l_{m_1+1}(y) \wedge \ldots \wedge f^l_{m_1+m_2}(x) = f^l_{m_1+m_2}(y)$, then $f^{nl}(x) = f^{nl}(y)$.

With this proposition, we can accept non-linear slicing functions of the above form; we just need to make sure that f is linearly independent of all the linear components.

6.2 Uniform Utilization

The UNIFORM UTILIZATION requirement asks the final coloring scheme P_h to uniformly utilize private components $g_1, \ldots, g_l$, for example the L1 and L2 caches. Previous work has proposed to compute P_h in two steps [16]. The first step computes the finest possible $P_{h'}$ that isolates shared resources (according to Equation (1)), and the second step obtains P_h from $P_{h'}$ by merging some colors to achieve uniform utilization (according to Equation (2)). Thus, even if we guarantee a certain number of colors in $P_{h'}$, some of those might be lost again during the merge in the second step. However, re-interpreting a proposition from [16], we can also enforce uniform utilization by choosing an appropriate indexing function for one of the shared resources:

Proposition 4. *If* $P_f \sqsubseteq P_h$ *and* $P_f \perp P_g$, *then* $P_h \perp P_g$.

This proposition shows that in order to make P_h uniformly utilize a private component g, it is enough that one of the shared resources f, for which $P_f \sqsubseteq P_h$ holds by construction, uniformly utilizes g. We can therefore use the algorithm presented in the previous subsection to synthesize f such that $P_f \perp P_{g_i}$ holds for all private components g_i.

Note that Proposition 4 is an implication, not an equivalence. Thus, enforcing uniform utilization of $g_1, \ldots, g_l$ while synthesizing f is sound, but there are cases in which we will not be able to find a solution on the level of f while there exists one for the coarser P_h.

7 Case Studies

We conduct three case studies, in which we synthesize alternative L3 set indexing functions. The first case study is an ablation study, in which we synthesize functions for increasingly demanding coloring requirements and observe the effect on performance. In the second case study, we synthesize L3 functions for a realistic server-class CPU, maxing out the number of colors the CPU supports. Lastly, we demonstrate that implementing the corresponding coloring scheme on top of one of our synthesized functions indeed eliminates cross-color interference on the colored components.

7.1 Implementation

We implemented our algorithms in Python in approx. 1.4K lines of code. The script receives a config file including:

- A list of functions $f_1, \ldots, f_l$ containing the indexing functions of other shared resources and architectural constraints. The list can be empty.
- A list of functions $g_1, \ldots, g_k$ containing the indexing functions of private components. The list can be empty.
- Potentially, a slicing function f_{slice}.
- A location containing the address traces of workloads for which we want to optimize performance.

Parameter	Value
Processor	1 core, FetchWidth=6, DecodeWidth=6, ExecWidth=4, RetireWidth=5, 352-entry ROB, 128-entry LQ, 72-entry SQ, 4 GHz frequency
L1 cache (I/D)	32KB, 2-way, 2-cycle latency
L2 cache	256KB, 4-way, 8-cycle latency
L3 cache	2MB, 16-way, 32-cycle latency
Prefetchers	disabled OR (L1I: Next-Line, L1D: Instruction-Pointer Stride, L2: Signature-Path)
Replacement	Least Recently Used (LRU)
DRAM	tRP=tRCD=tCAS=24, 16GB, open-row policy
Page size	4KB

Table 1: Configuration of the hardware simulated in the performance ablation study.

- A number 2^c of colors to be guaranteed.

If the script receives more than one function for the shared resources, it first computes a function f' with $P_{f'} = P_{f_1} \sqcup \ldots \sqcup P_{f_l}$ (as described in Section 5). Internally, we represent all functions as matrices and use the Sympy Python package for independence checks, basis computations, etc..

To construct the probability distribution from the address traces, according to Section 3.2, we would need to collect offsets in sliding windows the size of the number of resources. However, sliding windows of size 2048 (needed for typical L3 caches) for traces with millions of addresses are impractical to compute within a reasonable time frame. We experimented with smaller window lengths (between 1 and 30) and found that the impact on performance is below 0.5%. We therefore chose a fixed window length of 20 for our case studies.

7.2 Performance Ablation Study

First, we investigate how the performance of the synthesized functions changes depending on the complexity of the coloring requirements and the microarchitectural configurations. We are interested in the following research questions:

- **RQ1:** What is the performance of the synthesized functions compared to the default L3 indexing function?
- **RQ2:** How much does the hit rate decrease with increasingly complex coloring constraints?
- **RQ3:** Do prefetchers interfere with the performance of the synthesized functions?

7.2.1 Methodology

Simulator and Simulated Hardware. For the trace generation and evaluation of the indexing functions, we use ChampSim [13]. ChampSim is a trace-based simulator that is widely used in microarchitecture research and competitions to evaluate the performance of out-of-order cores and memory hierarchies [2, 4, 32, 38, 42–44]. The configuration of the simulated CPU is given in Table 1. We simulate a single-core CPU with a comparatively small L3 cache to increase the contention on the cache. Using a single-core CPU for this first case study enables us to conduct more experiments in a reasonable time frame. We move to a multi-core setup in the second case study. Prefetchers are disabled by default and only enabled in the experiment focused on prefetching.

Benchmarks. For address trace generation and evaluation, we use workloads from PARSEC 2.1, SPEC 2006, and SPEC 2017 (59 different workloads in total). We obtained the ChampSim traces for these workloads from the DPC-3 data prefetching competition [2] for SPEC and from the artifact of recent work by Bera et al. [4, 15] for PARSEC. For the address traces required by our tool, we adapt ChampSim to log all L3 accesses. Each workload is simulated for 150M instructions, where the first 50M serve as warm-up.

Experiments. We conduct five experiments, each time synthesizing L3 set indexing functions.

- **EXP1: Unconstrained Synthesis.** We synthesize L3 functions without coloring constraints and compare their performance to the L3 default.
- **EXP2: Enabling Prefetching.** We investigate if using prefetchers changes the difference in performance between the synthesized functions and the default function. Again, no coloring constraints are applied.
- **EXP3: Architectural Constraints Only.** We synthesize L3 set indexing functions such that the L3 cache can be colored with 4K pages.
- **EXP4: Architectural Constraints + DRAM.** We extend the previous experiment and additionally require that the L3 cache can be colored simultaneously with a DRAM module indexed by a typical bank indexing function.
- **EXP5: Architectural Constraints + DRAM + Slicing.** In addition to the previous experiment, the L3 cache is now sliced by a typical slicing function.

For the last three experiments, we synthesize functions up to the maximum number of colors that are theoretically possible for the given functions.

Training and Evaluation. All experiments are repeated ten times. In each round, we synthesize the function from the address traces of six randomly picked workloads and evaluate it on three different, randomly picked workloads. We report L3 hit rate, L3 miss latency, and number of instructions per cycle (IPC) across all ten rounds by the geometric mean of their relative gain (or overhead) over the default L3 set indexing function. For each benchmark, we first divide the metric of the synthesized function by the corresponding value of the default L3 function. Then, we take the geometric mean over all measurements and subtract 1, so that the baseline lies at 0. Hence, any reported number above (below) 0 corresponds to a relative gain (overhead) over the default L3. Note that for hit rate and IPC, a positive value indicates an improvement over the default L3 function, while for miss latency, it is the

$$
\begin{aligned}
f_0 &= x^{30} \oplus x^{29} \oplus x^{27} \oplus x^{25} \oplus x^{24} \oplus x^{17} \\
f_1 &= x^{31} \oplus x^{30} \oplus x^{29} \oplus x^{26} \oplus x^{24} \oplus x^{23} \oplus x^{19} \oplus x^{18} \oplus x^{15} \\
f_2 &= x^{24} \oplus x^{20} \oplus x^{19} \oplus x^{16} \oplus x^{14} \\
f_3 &= x^{31} \oplus x^{30} \oplus x^{28} \oplus x^{26} \oplus x^{24} \oplus x^{23} \oplus x^{22} \oplus x^{21} \oplus x^{16} \oplus x^{13} \\
f_4 &= x^{30} \oplus x^{26} \oplus x^{25} \oplus x^{24} \oplus x^{22} \oplus x^{21} \oplus x^{20} \oplus x^{12} \\
f_5 &= x^{31} \oplus x^{28} \oplus x^{27} \oplus x^{25} \oplus x^{24} \oplus x^{18} \oplus x^{16} \oplus x^{11} \\
f_6 &= x^{31} \oplus x^{29} \oplus x^{28} \oplus x^{26} \oplus x^{23} \oplus x^{22} \oplus x^{21} \oplus x^{20} \oplus x^{19} \oplus x^{18} \oplus x^{10} \\
f_7 &= x^{31} \oplus x^{30} \oplus x^{29} \oplus x^{28} \oplus x^{27} \oplus x^{26} \oplus x^{25} \oplus x^{24} \oplus x^{21} \oplus x^{20} \oplus \\
&\quad x^{19} \oplus x^{18} \oplus x^{16} \oplus x^{9} \\
f_8 &= x^{31} \oplus x^{29} \oplus x^{28} \oplus x^{24} \oplus x^{23} \oplus x^{22} \oplus x^{20} \oplus x^{19} \oplus x^{8} \\
f_9 &= x^{31} \oplus x^{28} \oplus x^{25} \oplus x^{22} \oplus x^{16} \oplus x^{7} \\
f_{10} &= x^{31} \oplus x^{26} \oplus x^{24} \oplus x^{23} \oplus x^{22} \oplus x^{21} \oplus x^{20} \oplus x^{18} \oplus x^{6}
\end{aligned}
$$

Figure 4: Synthesized L3 set indexing function without constraints. Each f_i defines one output bit.

opposite. Detailed numbers for each round can be found in the artifact accompanying this paper [10]. As a baseline, we use the default L3 indexing function that projects to bits 6 to 16. In each round, we evaluate the baseline function on the same three workloads as the synthesized function. In all experiments, we do not evaluate the performance impact of introducing page coloring to the system, but isolate the effect of replacing the L3 indexing function.

7.2.2 EXP1: Unconstrained Synthesis

Towards **RQ1**, we synthesize functions without further constraints. We observe that the performance is on par with the default L3 function. The absolute value of the gain over the default L3 is below 0.5% for hit rate, miss latency, and IPC.

Characteristics of the Functions. As an example, one of the synthesized functions is depicted in Figure 4. We observe that the function uses long XOR-chains and many of the more significant bits, similar to modern linear slicing functions [12, 17, 28]. We also see that most of the output bits contains exactly one of the eleven least significant non-block-offset bits. That way, addresses of linear strides will be mapped to different cache sets. Our synthesized functions usually have this shape: many XORs and lower-significant bits appear in exactly one of the output bits. All functions synthesized for our case studies can be found in the artifact [10].

7.2.3 EXP2: Enabling Prefetching

To answer **RQ3**, we enable typical prefetchers (see Table 1) and compare the performance of the default L3 function with the one synthesized in **EXP1** (for all ten batches of workloads). Prefetching does not make a difference: The hit rate, L3 miss latency, and IPC overheads are negligible.

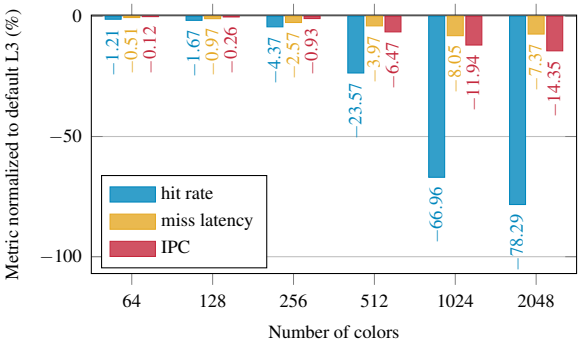


Figure 5: **EXP3**: Arch. Constraints Only.

7.2.4 EXP3: Arch. Constraints Only

To start answering **RQ2**, we synthesize functions that enable coloring the L3 cache with 4K pages. The default L3 function already supports 32 colors, so we start from 64. The maximum number of colors is 2048, which is the number of cache sets. The results of the experiments are depicted in Figure 5. The performance of the synthesized functions in terms of hit rate and IPC starts to degrade once we enforce 512 colors. This makes sense: the more colors we enforce, the more of the lower-significant bits have to be removed from the functions, as these bits do not appear in the 4K page indexing function. The functions for 64 colors do not use bit 11, the functions for 128 colors also remove bit 10, and so on. At 2048 colors, the functions only uses bits 12 and above, which is bad for access patterns like linear strides.

On the other hand, the L3 miss latency decreases. In this single-core experiment, DRAM contention arises only intra-core. The default DRAM mapping of ChampSim operates at page granularity, while the L3 indexing function drops lower address bits as the number of colors increases. Thus, the L3 miss stream exhibits higher spatial locality within 4KB pages, leading to increased row-buffer locality under this DRAM mapping. For instance, in the configuration that enforces 1024 (2048) colors, the row buffer hit rate increases by 31% (25%).

7.2.5 EXP4: Arch. Constraints + DRAM

We now synthesize an L3 function such that the L3 cache and DRAM banks can be partitioned simultaneously. Thus, the resulting coloring scheme is obtained by computing $P_h = P_f \sqcup P_{f_{4K}} \sqcup P_{f_{DRAM}}$. As bank indexing function, we choose a function that was recently reverse-engineered from an Intel i7-11700 machine from the RocketLake family [8]. As this function has four output bits, we can enforce at most 16 colors. The results are given in Figure 6, again with the default L3

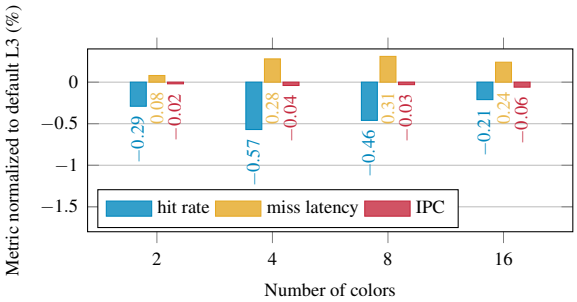


Figure 6: **EXP4**: Arch. Constraints + DRAM.

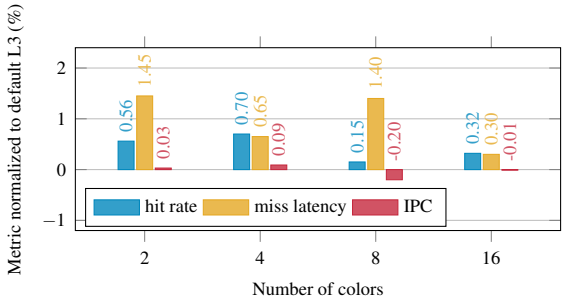


Figure 7: **EXP5**: Arch. Constraints + DRAM + Slicing.

indexing function as baseline. The lies within the normal variance also observed in **EXP1**.

7.2.6 EXP5: Arch. Constraints + DRAM + Slicing

Lastly, we extend the L3 cache with a linear slicing function, meaning that the coloring scheme is now defined as $P_h = P_{(f_{slice},f)} \sqcup P_{f_{4K}} \sqcup P_{f_{DRAM}}$. We use the slicing function of the same Intel i7-11700 machine [28]. Since the cache is sliced now, the number of cache sets increases when keeping the dimensions of the L3 set indexing function. We therefore reduce the number of ways accordingly to keep the size of the cache the same. As baseline, we now choose a cache sliced with the same slicing function but with the default L3 set indexing function. The results are depicted in Figure 7. Again, this setting permits up to sixteen colors, and there is no noticeable performance drop.

7.3 Performance on a Server-Class CPU

In the second case study, we evaluate our approach on a server-class CPU. We aim to answer **RQ4:** Can we increase the number of colors on a realistic server-class CPU and what is the performance of the synthesized set indexing function?

7.3.1 Methodology

Simulated Hardware and Benchmarks. We simulate a Intel Xeon Gold 6346 processor [3] in ChampSim, a 16-core server-class CPU whose microarchitecture belongs to the Ice Lake generation.[1] The ChampSim configuration is given in Table 2. We use the same benchmarks as before. Each core executes one workload, first 50M instructions to warm up, then for at least 50M further instructions. If a core finishes early, it continues executing until every core has finished.

Parameter	Value
Processor	16 cores, FetchWidth=6, DecodeWidth=6, ExecWidth=4, RetireWidth=5, 352-entry ROB, 128-entry LQ, 72-entry SQ, 4 GHz frequency
L1 cache (I/D)	(per core) 32KB, 8-way, 2-cycle latency
L2 cache	(per core) 1MB, 16-way, 8-cycle latency
L3 cache	(shared) 36MB, 12-way, 32-cycle latency, 24 slices
Prefetchers	disabled
Replacement	Least Recently Used (LRU)
DRAM	tRP=tRCD=tCAS=24, 8 channels, 512GB, open-row policy
Page size	4KB and 2MB

Table 2: Hardware configuration used in multi-core server-class case studies.

Experiment. We aim to synthesize an L3 set indexing function that maximizes the number of colors for a realistic DRAM bank indexing and L3 slicing function (**EXP6**). The slicing function of Xeon Gold 6346 processors has been reverse-engineered recently [28]. It has 24 slices, i.e., the slicing function has three linear bits and two non-linear bits. The non-linear bits have the shape described in Section 6.1.1; we can thus use our approach to synthesize a set indexing function that can be hierarchically composed with this slicing function. The DRAM bank indexing function of this specific processor does not seem to be publicly known when finalizing this paper. We thus use a recently reverse-engineered bank indexing function of an Intel Xeon Gold 5318Y processor [25], which also belongs to the Ice Lake generation:

$$f_{DRAM}(x) = x^{37}, x^{36}, x^{35}, x^{16}, x^{6} \oplus x^{24}, x^{21} \oplus x^{25}, x^{22} \oplus x^{26}, x^{23} \oplus x^{27}, x^{8} \oplus x^{14} \oplus x^{26}, x^{9} \oplus x^{15} \oplus x^{27}, x^{11} \oplus x^{14} \oplus x^{17} \oplus x^{25} \oplus x^{26}$$

The first three higher bits index the channels. We implemented this custom channel and bank indexing in ChampSim. In this study, we enforce coloring with 2MB pages instead of 4KB.

Training and Evaluation. Due to the increased simulation time, we repeat the experiment only three times for all configurations. We synthesize the function from 32 randomly picked workloads and evaluate it by simulating the CPU with 16 randomly chosen different workloads, one for each core. The training workloads are run in a single-core setup with

[1] We do not simulate higher core counts due to the substantial turnaround time of running the full experimental suite. Importantly, this limitation does not stem from any scalability constraint of our proposed approach.

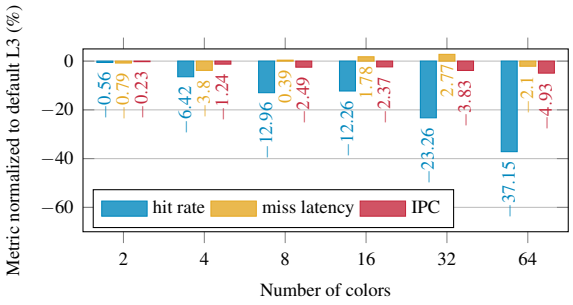


Figure 8: **EXP6**: Coloring Xeon Gold 6346 with 2MB pages.

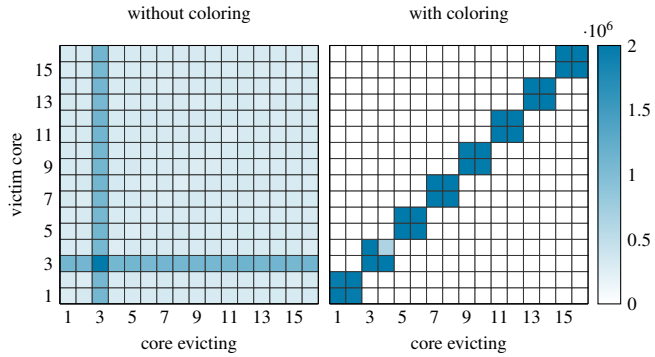


Figure 9: L3 evictions with and without coloring. The x-axis depicts the core causing the eviction of a cache line, the y-axis depicts the core that initially accessed the evicted line. The gradient indicates the number of observed evictions during a simulation of 50M instructions.

one slice of L3 cache (i.e., 1.5MB) to obtain the required L3 address traces. The single-core setup is preferred for collecting traces, also in the multi-core setup, to avoid cross-core access offsets polluting the offset probability distribution. As a baseline, we again use the default L3 set indexing function. The gain in terms of hit rate is calculated as in the previous experiments. As the IPC and L3 miss latency are measured per core, we treat them as individual measurements: we divide the per-core value from the synthesized functions by the per-core value from the default L3, then take the geometric mean of the relative gain/overhead across all cores and all three iterations.

7.3.2 Results

With the default L3 set indexing function, the processor only supports a single color, i.e., it is not suitable for securely partitioning the system. With our algorithms, we can reach up to 64 colors by synthesizing alternative L3 set indexing functions. Figure 8 presents the performance metrics. We see that the more colors we enforce, the more pronounced is the drop in the hit rate. If we were to partition the CPU with 16 colors (i.e., each trust domain would get its own core), there would be a mean overhead in the hit rate of 12.26% relative to the default L3 set indexing function. In absolute terms, this corresponds to an average reduction in hit rate of 5.5%. For 16 colors, one of the three functions and the corresponding coloring function h are as follows:

$$f_{L3}(x) = x^{35}, x^{27} \oplus x^{23}, x^{26} \oplus x^{22}, x^{25} \oplus x^{21}, x^{16}, x^{15} \oplus x^{14}, x^{13}, x^{12}, x^{11}, x^{15} \oplus x^{10}, x^{15} \oplus x^{9}$$
$$h(x) = x^{35}, x^{27} \oplus x^{23}, x^{26} \oplus x^{22}, x^{25} \oplus x^{21}$$

As one can see, the set indexing function keeps one of the channel bits for coloring. It also XORs many of the bits seen in the DRAM function. The function was designed to be used with a slicing function that uses many of the lower-significant bits, see [28].

On a Xeon Gold 6346 CPU, we can increase the number of colors from 1 to 64 by synthesizing an alternative L3 set indexing function.

7.4 Security Evaluation

We conduct a final case study to ensure that the coloring scheme we compute based on the synthesized function indeed provides effective microarchitectural isolation.

Methodology. We emulate an attacker priming the L3 cache by implementing a simple C program that walks through 512MB of memory. We trace the program using the pin tracer [1] and run it on one of the 16 cores of the multi-core configuration, see Table 2. The other cores run one of the SPEC benchmarks. In the baseline system, we again use the Intel Xeon Gold 5318Y DRAM function (as in **EXP6**) and the default L3 function. For both L3 and DRAM row buffers, we record the evicting and victim cores of evicted cache lines and rows, respectively. For a non-colored system, we expect the adversarial core to evict cache lines and rows from all other cores. For the colored-system, we implement the synthesized L3 function obtained in the 8-color experiment (Figure 8) and assign to each pair of cores the memory pages of one color, based on the corresponding coloring scheme. The DRAM function stays the same. We expect evictions to be confined to the attacker's color.

Results. Figure 9 depicts the L3 evictions with and without coloring. The adversarial program runs on core 3. Without coloring, evictions are quite evenly spread across cores. With coloring, there are no cross-color evictions anymore. Figure 10 depicts the DRAM row evictions without and with coloring. Compared to the L3 cache, the accesses of the adversarial

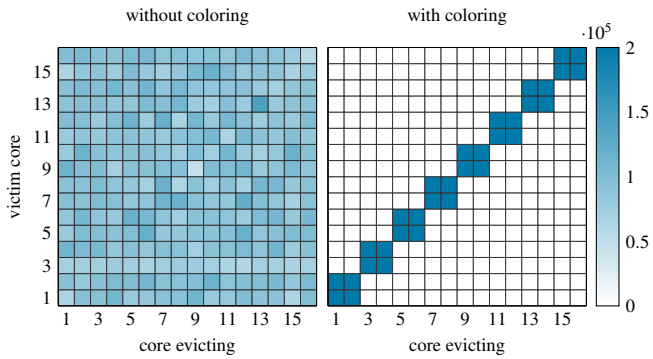


Figure 10: DRAM evictions with and without coloring. Labels are as in Figure 9.

core are not that easy to distinguish anymore. However, in the colored variant, we still see that cores can only cause evictions of cores belonging to the same color. Note that in CPUs (e.g, Intel) where the coherence directory has the same indexing function as the L3 cache [16, 45], the coloring scheme would also prevent directory-based attacks.

Multi-component memory coloring prevents cross-color evictions in both the L3 cache and DRAM banks.

8 Discussion

Applicability of our Approach. Memory coloring is known to be an effective technique to eliminate side channels on shared microarchitectural components. It has been applied to partition caches [11, 34], DRAM [39, 46], and coherence directories [39]. Memory coloring has also been shown to be effective against Rowhammer attacks by coloring DRAM subarrays [23]. This work aims to make multi-component coloring applicable by synthesizing alternative indexing functions that increase the number of colors a machine supports. Our approach is applicable to all microarchitectural components that are physically indexed by a linear function (e.g., shared caches, DRAM banks, coherence directories).

Remaining Leakage. Our approach is targeted at cross-core side channels occurring in a cloud setting; it therefore does not prevent side channels via on-core caches and buffers, e.g., the branch prediction unit. Additionally, spatial partitioning does not prevent utilization-based side-channels, where timing is impacted by how congested the resource is. Further information leakage might occur via the shared hypervisor context, e.g., when performing I/O and hypercalls, or via interrupt handlers.

Latency. The latency of XOR gates is small compared to the latency induced by (L3) cache hits and misses. The gate depth is logarithmic in the number of XORs per output bit. In the worst case, we XOR 48 bits, resulting in a depth of 6. Assuming a cost of 2 FO4 per XOR, our functions do not cause more than 2 cycles delay, even in high-frequency cores. Still, one might consider bounding the number of XORs algorithmically to reduce latency. This is not trivial, as our algorithms do not synthesize the function directly, but rather the partition it defines. This is left for future work.

Further Future Work. We only studied the case of synthesizing a single indexing function (the L3 set indexing function in our case studies), assuming the other indexing functions to be fixed. A CPU designer that is free to choose all indexing functions might want to synthesize several functions in parallel to obtain even more colors while maximizing overall system performance. Secondly, we have considered non-linear functions only in the case of slicing functions, and only if they take the shape of feeding the result of linear functions into non-linear mixers. It is unclear how to color with general non-linear functions and also how to synthesize non-linear indexing functions.

9 Related Work

Softare-Based Memory Coloring. We use software-based memory coloring, where the memory manager of the host OS or the hypervisor implement the coloring scheme. While our focus is on security, early use cases of memory coloring targeted single-tenant performance and workload predictability [7, 19, 29, 41]. Most other software-based coloring techniques only partition one component at a time [11, 34, 46]. There are a few works that discuss multi-component partitioning [16, 36, 39], but they do not synthesize indexing functions to improve the number of colors.

Beyond Software-Based Memory Coloring. Besides memory coloring, CAT is a hardware-based solution for partitioning caches. It partitions the L3 cache along its ways [33]. CAT can be combined with software-based techniques, but the number of colors is naturally limited to the number of ways of the cache. There also exist alternative cache designs that support partitioning [40]. Besides partitioning, information leakage via caches can be prevented by randomized cache replacement policies [30, 40], encrypted cache indexing [26, 27], or by locking L3 cache lines [21].

Indexing Function Design. The design of indexing functions has received significantly less attention than the design of replacement policies or prefetchers. The discussion on indexing function design is dominated by performance and computational latency. Since the late 1990s, it is known that cache indexing functions that just project on m bits from the address

to index the cache set are not robust and susceptible to repetitive conflict misses [5,37]. Linear cache slicing functions thus often use significantly more address bits [12,17,28]. Alternatively, one uses indexing functions that are (at least partially) non-linear, e.g., by using prime modulo operations [9,20] or by combining XOR chains with small non-linear mixer circuits [28]. The downside of these approaches is the increased latency of the required computations [9,43].

Related to our work, there are a few approaches that synthesize indexing functions to increase the performance for a given set of workloads [42–44]. They suggest to dynamically adapt the indexing function depending on the executed workload, for example by observing address bits with the highest perceived randomness [43] or by employing reinforcement learning [44].

10 Conclusion

Microarchitectural indexing functions should be designed with both security and performance requirements in mind. For memory coloring, in particular, the number of colors achievable on a CPU critically depends on the algebraic structure of the indexing functions, which limits the applicability of the technique. To address this limitation, we have presented algorithms that automatically synthesize indexing functions guaranteeing a minimum number of colors while, subject to this constraint, maximizing performance. Our approach furthermore ensures that the synthesized functions and the resulting coloring scheme are compatible with the surrounding system design. Our case study shows that alternative L3 cache indexing functions can indeed increase the number of colors in multi-core CPUs.

Acknowledgements. We thank the anonymous USENIX Security reviewers for their insightful suggestions, which had a great impact on the paper. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

Ethical Considerations

This work proposes novel algorithms to synthesize indexing functions for microarchitectural components to achieve microarchitectural isolation through memory coloring. It supports the development of principled system-level protections against side channel attacks. It does not include any offensive techniques, i.e., no attacks nor exploits nor reverse-engineered results. It does not involve human subjects, user studies, or the collection of private or personally identifiable data. In the following, we carefully considered the ethical implications of our work based on a stakeholder-based ethics analysis.

Stakeholders. The primary stakeholders affected by this work include CPU designers who may wish to use our algorithms for indexing function design to provide efficient microarchitectural isolation schemes. Furthermore, the broader research community, which may build upon our results, can be affected by our work.

Potential Harms & Mitigations. Our algorithms may influence future designs of indexing functions. There is a risk of our algorithms being used under wrong assumptions. Furthermore, their effectiveness depends on a correct implementation. To mitigate this risk, we are explicit about the isolation guarantees one can expect from the provided functions and the settings in which they hold. The major claims regarding the effectiveness of our algorithms rely on formal proofs that we provide in the appendix. We also provide additional functions in our publicly available artifact that allow any indexing function to be tested for its coloring and isolation properties. Thus, anyone can arrive at an informed decision before using the results of our work. With this in mind, we did not identify any ethical concerns for publicly sharing this work.

Open Science

This paper is accompanied by an artifact [10] containing:

- The implementations of all algorithms presented in this paper, and all the required files to execute them. Separate functions are provided to test any set of indexing functions for their joint isolation properties.
- All datasets we used in our evaluation, including address traces we used for training and evaluation. We also provide the probability distributions obtained from the training sets.
- The ChampSim simulator with our modifications to generate address traces and reproduce our results.
- Detailed instructions to run all provided software.

References

[1] Pin - a dynamic binary instrumentation tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html. Accessed on 24th May, 2026.

[2] The 3rd data prefetching championship, 2019. https://dpc3.compas.cs.stonybrook.edu. Accessed on 21st May, 2026.

[3] Intel xeon gold 6346 processor, 2019. https://www.intel.com/content/www/us/en/products/sku/212457/intel-xeon-gold-6346-processor-36m-cache-3-10-ghz/specifications.html. Accessed on 21st May, 2026.

[4] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021. https://doi.org/10.1145/3466752.3480114.

[5] François Bodin and André Seznec. Skewed associativity enhances performance predictability. In David A. Patterson, editor, *Proceedings of the 22nd Annual International Symposium on ComputerArchitecture, ISCA 1995*. ACM, 1995. https://doi.org/10.1145/223982.224437.

[6] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization (seventh printing with corrections)*. Cambridge, UK, Cambridge University Press, 2009.

[7] Edouard Bugnion, Jennifer-Ann M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. Compiler-directed page coloring for multiprocessors. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, 1996. https://doi.org/10.1145/237090.237195.

[8] Weijie Chen, Shan Tang, Yulin Tang, Xiapu Luo, Yinqian Zhang, and Weizhong Qiang. ρhammer: Reviving rowhammer attacks on new architectures via prefetching. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2025. https://doi.org/10.1145/3725843.3756042.

[9] Jeffrey R. Diamond, Donald S. Fussell, and Stephen W. Keckler. Arbitrary Modulus Indexing. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2014. http://ieeexplore.ieee.org/document/7011384/.

[10] Stephan Dübler, Jana Hofmann, Boris Köpf, and Stavros Volos. Principled design of indexing functions for memory coloring - artifact, February 2026. Available at https://doi.org/10.5281/zenodo.18494189.

[11] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: The missing os abstraction. In *Proceedings of the Fourteenth EuroSys Conference*. ACM, 2019. https://doi.org/10.1145/3302424.3303976.

[12] Lukas Gerlach, Simon Schwarz, Nicolas Faroß, and Michael Schwarz. Efficient and generic microarchitectural hash-function recovery. In *Symposium on Security and Privacy (S&P)*. IEEE, 2024. https://doi.org/10.1109/SP54263.2024.00028.

[13] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jiménez, Elvira Teran, Seth H. Pugsley, and Jinchun Kim. The championship simulator: Architectural simulation for education and competition. *CoRR*, abs/2210.14324, 2022. https://doi.org/10.48550/arXiv.2210.14324.

[14] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, 1966. https://ieeexplore.ieee.org/document/6767827.

[15] SAFARI Research Group. Parsec 2.1 traces for champsim, 2024. Available at https://doi.org/10.5281/zenodo.14268119. Accessed on 26t May 2026.

[16] Jana Hofmann, Cédric Fournet, Boris Köpf, and Stavros Volos. Gaussian Elimination of Side-Channels: Linear Algebra for Memory Coloring. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024. https://dl.acm.org/doi/10.1145/3658644.3690263.

[17] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *Euromicro Conference on Digital System Design, DSD*. IEEE Computer Society, 2015. https://doi.org/10.1109/DSD.2015.56.

[18] Richard M. Karp. *Reducibility among Combinatorial Problems*. Springer US, Boston, MA, 1972. https://doi.org/10.1007/978-1-4684-2001-2_9.

[19] Richard E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, 1992.

[20] Mazen Kharbutli, Keith Irwin, Yan Solihin, and Jaejin Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *10th International Conference on High-Performance Computer Architecture (HPCA)*. IEEE, 2004. https://doi.org/10.1109/HPCA.2004.10015.

[21] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *21th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2012. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kim.

[22] Richard E. Korf. A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 106(2):181–203, 1998. https://www.sciencedirect.com/science/article/pii/S0004370298000861.

[23] Kevin Loughlin, Jonah Rosenblum, Stefan Saroiu, Alec Wolman, Dimitrios Skarlatos, and Baris Kasikci. Siloz: Leveraging dram isolation domains to prevent inter-vm rowhammer. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery, 2023. https://doi.org/10.1145/3600006.3613143.

[24] Stephan Mertens. The Easiest Hard Problem: Number Partitioning. In *Computational Complexity and Statistical Physics*. Oxford University Press, December 2005. https://academic.oup.com/book/41797/chapter/354512698.

[25] Antoine Plin, Lorenzo Casalino, Thomas Rokicki, and Ruben Salvador. Knock-knock: Black-box, platform-agnostic dram address-mapping reverse engineering. In *2nd Microarchitecture Security Conference (uASC)*, 2026. https://uasc.cc/proceedings26/uasc26-plin.pdf.

[26] Moinuddin K. Qureshi. CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In *51st Annual International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2018. https://doi.org/10.1109/MICRO.2018.00068.

[27] Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. ACM, 2019. https://doi.org/10.1145/3307650.3322246.

[28] Mikka Rainer, Lorenz Hetterich, Fabian Thomas, Tristan Hornetz, Leon Trampert, Lukas Gerlach, and Michael Schwarz. Rapid reversing of non-linear CPU cache slice functions: Unlocking physical address leakage. In *Symposium on Security and Privacy (S&P)*. IEEE, 2025. https://doi.org/10.1109/SP61157.2025.00206.

[29] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the first ACM Cloud Computing Security Workshop (CCSW)*. ACM, 2009. https://dl.acm.org/doi/10.1145/1655008.1655019.

[30] Gururaj Saileshwar and Moinuddin K. Qureshi. MIRAGE: mitigating conflict-based cache attacks with a practical fully-associative design. In *30th USENIX Security Symposium, (USENIX Security)*. USENIX Association, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/saileshwar.

[31] Ethan L. Schreiber, Richard E. Korf, and Michael D. Moffitt. Optimal Multi-Way Number Partitioning. *Journal of the ACM*, 65(4):1–61, August 2018. https://dl.acm.org/doi/10.1145/3184400.

[32] Ishan Shah, Akanksha Jain, and Calvin Lin. Effective mimicry of belady's MIN policy. In *IEEE International Symposium on High-Performance Computer Architecture, (HPCA)*. IEEE, 2022. https://doi.org/10.1109/HPCA53966.2022.00048.

[33] Mohammad Shahrad, Sameh Elnikety, and Ricardo Bianchini. Provisioning differentiated last-level cache allocations to vms in public clouds. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. Association for Computing Machinery, 2021. https://doi.org/10.1145/3472883.3487006.

[34] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE/IFIP, 2011. https://doi.org/10.1109/DSNW.2011.5958812.

[35] Carl P Simon and Lawrence Blume. *Mathematics for Economists*. Norton & Company, New York, 1994.

[36] Noriaki Suzuki, Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *16th International Conference on Computational Science and Engineering (ICCS)*. IEEE, 2013. https://doi.org/10.1109/CSE.2013.106.

[37] Nigel P. Topham, Antonio González, and José González. The design and performance of a conflict-avoiding cache. In Mark Smotherman and Tom Conte, editors, *Proceedings of the Thirtieth Annual International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 1997. https://doi.org/10.1109/MICRO.1997.645799.

[38] Georgios Vavouliotis, Gino Chacon, Lluc Alvarez, Paul V. Gratz, Daniel A. Jiménez, and Marc Casas. Page size aware cache prefetching. In *55th International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2022. https://doi.org/10.1109/MICRO56248.2022.00070.

[39] Stavros Volos, Cédric Fournet, Jana Hofmann, Boris Köpf, and Oleksii Oleksenko. Principled microarchitectural isolation on cloud cpus. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024. https://doi.org/10.1145/3658644.3690183.

[40] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *34th International Symposium on Computer Architecture (ISCA)*. ACM, 2007. https://doi.org/10.1145/1250662.1250723.

[41] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Making shared caches more predictable on multicore platforms. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2013. https://doi.org/10.1109/ECRTS.2013.26.

[42] Kevin Weston, Vahid Janfaza, Avery Johnson, and Abdullah Muzahid. A cost-effective dueling framework for set-associative cache indexing. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS)*, 2025. https://doi.org/10.1145/3721145.3729513.

[43] Kevin Weston, Avery Johnson, Vahid Janfaza, Farabi Mahmud, and Abdullah Muzahid. Customizing Cache Indexing Through Entropy Estimation. *57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024. https://ieeexplore.ieee.org/document/10764673/.

[44] Kevin Weston, Farabi Mahmud, Vahid Janfaza, and Abdullah Muzahid. SmartIndex: Learning to Index Caches to Improve Performance. *IEEE Computer Architecture Letters*, 22(1):33–36, January 2023. https://ieeexplore.ieee.org/document/10092937/.

[45] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE Symposium on Security and Privacy, (S&P)*, 2019. https://doi.org/10.1109/SP.2019.00004.

[46] Heechul Yun, Renato Mancuso, Zheng Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014. https://doi.org/10.1109/RTAS.2014.6925999.

A Collision Optimization Problem

In this section, we revisit the optimization goal given in Section 3.2. In order to obtain the general optimum in terms of values $M_0, M_1, \ldots, M_{2^m-1}$, we abstract from the goal of finding a linear function realizing them, but focus on the values itself. Further, since all offsets are mapped to some class, we also need to consider the condition $M_0 + \sum_{i=1}^{2^m-1} M_i = 1$. Therefore, we attain the following optimization problem:

Optimization Problem: Find $M_0, M_1, \ldots M_{2^m-1} \in [0,1]$ *for:*

$$\max_{M_0, M_1, \ldots, M_{2^m-1}} \quad p(\neg Miss) = (2^m-1)! \cdot M_1 \cdot M_2 \cdot \ldots \cdot M_{2^m-1}$$
$$\textit{subject to} \quad M_0 + \sum_{i=1}^{2^m-1} M_i = 1 \tag{3}$$

Since we want to make use of all available resources, it is reasonable to assume $\forall i \in \{1, \ldots, 2^m-1\} : M_i \neq 0$. This is a concave optimization problem with an equality constraint, a known class of optimization problems. This is especially true when considering $-p(\neg\text{Miss})$ as the target function, which turns it into a convex optimization problem with an equality constraint. For this class of optimization problems, Lagrangian Optimization is a well-known and often-used solution technique [35, Th. 18.1], [6, Ch. 10], which we will apply in the following. To start with, we set up the Lagrangian with Lagrange multiplier λ:

$$L := (2^m-1)! \cdot M_1 \cdot M_2 \cdot \ldots \cdot M_{2^m-1} + \lambda(1 - (M_0 + \sum_{i=1}^{2^m-1} M_i))$$

Whenever we have found an optimum $(M_1, M_2, ..., M_{2^m-1}, \lambda)$ for L, we have also found an optimum $(M_1, M_2, ..., M_{2^m-1})$ for $p(\neg\text{Miss})$. The remaining value M_0 can then be calculated by substitution and partial differentiation of $p(\neg\text{Miss})$ w.r.t. M_0. To simplify notation, we denote a product sequence where just one item is missing as

$$M_1 \cdot \ldots \cdot \widehat{M_i} \cdot \ldots \cdot M_{2^m-1} := M_1 \cdot \ldots \cdot M_{i-1} \cdot M_{i+1} \ldots \cdot M_{2^m-1}.$$

In order to maximize L, we differentiate by all M_i, leading to

$$\forall i \in \{1, \ldots, 2^m-1\} :$$
$$\frac{\partial L}{\partial M_i} = (2^m-1)! \cdot M_1 \cdot \ldots \cdot \widehat{M_i} \cdot \ldots \cdot M_{2^m-1} - \lambda \stackrel{!}{=} 0 \tag{4}$$

and by λ, which gives us the partial derivative

$$\frac{\partial L}{\partial \lambda} = 1 - (M_0 + \sum_{i=1}^{2^m-1} M_i) \stackrel{!}{=} 0. \tag{5}$$

From Equation (4) we get:

$$\forall i \in \{1, \ldots, 2^m-1\} : M_1 \cdot \ldots \cdot \widehat{M_i} \cdot \ldots \cdot M_{2^m-1} = \frac{\lambda}{(2^m-1)!}$$
$$\Rightarrow \forall i, j \in \{1, \ldots, 2^m-1\} :$$
$$M_1 \cdot \ldots \cdot \widehat{M_i} \cdot \ldots \cdot M_{2^m-1} = M_1 \cdot \ldots \cdot \widehat{M_j} \cdot \ldots \cdot M_{2^m-1}$$

Since $\forall i \in \{1, \ldots, 2^m-1\} : M_i \neq 0$, we can factor out all M_ι with $\iota \neq i$ and $\iota \neq j$ which implies $\forall i, j \in \{1, \ldots, 2^m-1\} : M_i = M_j$. We denote $M := M_1 = M_2 = \ldots = M_{2^m-1}$. Substituting all M_i in Equation (5) with M results in

$$M_0 + (2^m-1)M = 1 \Leftrightarrow M = \frac{1-M_0}{2^m-1}. \tag{6}$$

For deriving the optimal M_0 we insert Equation (6) in the original target function from (3)

$$p(\neg\text{Miss}) = (2^m-1)!\left(\frac{1-M_0}{2^m-1}\right)^{2^m-1}$$

and differentiate by M_0:

$$\begin{aligned}\frac{\partial p(\neg\text{Miss})}{\partial M_0} &= (2^m-1)!(2^m-1)\left(\frac{1-M_0}{2^m-1}\right)^{2^m-2}\left(\frac{-1}{2^m-1}\right)\\ &= -(2^m-1)!\left(\frac{1-M_0}{2^m-1}\right)^{2^m-2}\end{aligned}$$

It holds $\left(\frac{1-M_0}{2^m-1}\right)^{2^m-2} > 0$, since $M_0 \leq 1$ and also since 2^m-2 is always even, as well as $(2^m-1)! > 0$. Therefore the partial derivative of $p(\neg\text{Miss})$ w.r.t. M_0 is negative for all values of M_0:

$$\frac{\partial p(\neg\text{Miss})}{\partial M_0} < 0$$

This means that for maximizing $p(\neg\text{Miss})$ under the given constraint, the value M_0 should be chosen as small as possible, i.e., $M_0 = 0$. At the same time, all M_i should be equal, i.e., $M_1 = M_2 = \ldots = M_{2^m-1} = \frac{1}{2^m-1}$.

B Kernel Requirement for Memory Coloring

In this section, we prove Proposition 1. Let the functions $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ and $g : \mathbb{F}_2^n \to \mathbb{F}_2^{m'}$ be given. As a first step, we write $P_f \sqcup P_g$ as a factor space

$$P_f \sqcup P_g = \mathbb{F}_2^n / ker(f) + ker(g),$$

which implies

$$|P_f \sqcup P_g| = \frac{|\mathbb{F}_2^n|}{|ker(f)+ker(g)|}. \tag{7}$$

The kernel of any linear function is also a vector space, and so the *dimension* of $P_f \sqcup P_g$ is well-defined. The dimension of a vector space is given by the number of linearly independent vectors, e.g. the dimension of $\mathbb{F}_2^2$ would be 2, written $dim\left(\mathbb{F}_2^2\right) = 2$. We can dissect

$$\begin{aligned}&dim\left(ker(f)+ker(g)\right)\\ &= dim\left(ker(f)\right)+dim\left(ker(g)\right)-dim\left(ker(f)\cap ker(g)\right).\end{aligned}$$

This gives us a direct handle for the number of classes in $P_f \sqcup P_g$, since $dim\left(ker(g)\right) = n-m'$ and $dim\left(ker(f)\right) = n-m$ is given by the dimensions of the spaces they operate between $(\mathbb{F}_2^n, \mathbb{F}_2^m, \mathbb{F}_2^{m'})$. When 2^c many classes are desired for $P_f \sqcup P_g$, i.e. $|P_f \sqcup P_g| = 2^c$, we can rearrange Equation (7) and hence need to satisfy

$$|ker(f)+ker(g)| = \frac{|\mathbb{F}_2^n|}{|P_f \sqcup P_g|} = \frac{2^n}{2^c} = 2^{n-c}.$$

Since $|ker(f)+ker(g)| = 2^{dim(ker(f)+ker(g))}$, it is required that $dim\left(ker(f)+ker(g)\right) = n-c$ holds. We have

$$\begin{aligned}dim\left(ker(f)+ker(g)\right) =& dim\left(ker(f)\right)+dim\left(ker(g)\right)\\ &- dim\left(ker(f)\cap ker(g)\right)\end{aligned}$$

and so we arrive at

$$\begin{aligned}&dim\left(ker(f)\cap ker(g)\right)\\ &= dim\left(ker(f)\right)+dim\left(ker(g)\right)-dim\left(ker(f)+ker(g)\right)\\ &\overset{!}{=} (n-m)+(n-m')-(n-c) = n-m'-m+c.\end{aligned}$$

For our algorithm, this means that we need to ensure that sufficiently many items from $ker(g)$, which can be calculated from g, are included in $ker(f)$.

C Kernel Requirement for Uniform Utilization

In this section, we prove Proposition 2. We start by citing the following proposition [16, Prop. 3]:

Proposition 5. *Let f, g be linear functions defined on $\mathbb{F}_2^n$. Then the following are equivalent*

1. $P_f \perp P_g$
2. $rank\left(\begin{bmatrix} A_f \\ A_g \end{bmatrix}\right) = rank\left(A_f\right)+rank\left(A_g\right)$

In our context, we have

$$\begin{aligned}&P_{f_{slice}} \perp P_f\\ &\equiv\ rank\left(\begin{bmatrix} A_{f_{slice}} \\ A_f \end{bmatrix}\right) = rank\left(A_{f_{slice}}\right)+rank\left(A_f\right).\end{aligned}$$

This means that none of the rows in A_f are linearly dependent from the rows in $A_{f_{slice}}$, or in other words: The rowspaces of f_{slice} and f do not overlap. From this it follows:

$$\begin{aligned}P_f \perp P_g \ &\equiv\ rowspace(f_{slice}) \cap rowspace(f) = 0\\ &\equiv\ \left(rowspace(f_{slice}) \cap rowspace(f)\right)^{\perp} = \mathbb{F}_2^n\\ &\equiv\ ker(f_{slice})+ker(f) = \mathbb{F}_2^n\end{aligned}$$

Therefore, $ker(f)$ needs to cover all of $\mathbb{F}_2^n$ not occupied by $ker(f_{slice})$. This means that for a function f with $rowspace(f) \cap ker(f) = 0$, the rowspace of f_{slice} needs to be in $ker(f)$. So we arrive at the condition

$$rowspace(f_{slice}) \subseteq ker(f) \ \wedge\ rowspace(f) \cap ker(f) = 0.$$