Understanding Compressed Caching through the Compressed Optimal

Michael “Tres” Brenan III
Department of Computer Science
The University of Texas at Austin
msbrenan@cs.utexas.edu

May 10th, 2019
Abstract

Cache sizes have expanded dramatically in recent years to meet the demands of modern memory-intensive applications; however, further increasing on-chip cache sizes has become more difficult due to complex tradeoffs between die space, latency, and power. Hardware cache compression is an attractive alternative to directly increasing the cache size, as it allows for higher effective cache capacity - and thus better cache hit rates and system performance - while keeping the physical size of the cache the same. While prior work has demonstrated the potential performance benefits of employing compression in last-level caches through experimentation, none have answered the question: what is the potential – in terms of improved hit rate – for compressed caches? This thesis answers this question by providing a tractable integer linear programming formulation of the compressed optimal hitrate, and evaluates the optimal hit rate of several compressed caching architectures. In doing so, we show that more restrictive, practical architectures like Yet Another Compressed Cache (YACC) can achieve similar theoretical hit rates to more permissive architectures; and furthermore, that there is a large gap between the compressed optimal and existing compressed replacement policies like CAMP. Finally, to try and close this gap, we propose an extension of the Hawkeye replacement policy for the YACC compressed cache. On 22 SPEC2006 memory-intensive benchmarks, our Hawkeye variant obtains an 18% higher hitrate than CAMP on average (13% higher on compressible benchmarks); and a 13% higher hitrate than YACC+RRIP on average (10% higher on compressible benchmarks). This translates to Compressed Hawkeye having 3.24% higher IPC than CAMP on average (and up to 12% higher); and 2.30% higher IPC than YACC+RRIP on average (up to 12% higher).
## Contents

1 Introduction ......................................................... 2

2 Background .......................................................... 5
   2.1 Cache-Line Compression: CPACK+Z ............................. 5
   2.2 Cache-Line Storage: A Simple Classification ................. 6
   2.3 Cache-Line Replacement: YACC and CAMP .................... 7

3 The Compressed Optimal ............................................. 10
   3.1 Belady’s Algorithm and Usage Intervals ....................... 10
   3.2 Belady’s Algorithm for Compressed Caching .................. 11
   3.3 Computing the Compressed Optimal ............................ 13
      3.3.1 Heterogeneous Optimal .................................. 14
      3.3.2 Homogeneous Optimal ................................... 15
      3.3.3 Superblock-Homogeneous Optimal ......................... 16
      3.3.4 Other Methods ........................................... 17
   3.4 Empirical Results on Benchmarks .............................. 18

4 Compressed Hawkeye ................................................. 26
   4.1 Empirical Results .............................................. 28

5 Conclusion & Acknowledgements .................................. 31
   5.1 Future Work ..................................................... 31
   5.2 Conclusion ....................................................... 31
   5.3 Acknowledgements ............................................... 32
Chapter 1

Introduction

As processors add more cores and applications become more memory-intensive, system designers have increasingly struggled with the so called “memory wall” [3, 26]: the large disparity between core performance and main memory latency/bandwidth. This disparity has become so severe that 800-1,000 integer add operations (0.25ns per 4 operations) can be completed in the same amount of time as fetching a single cache line from main memory (100ns per operation) [18]. Many approaches have been taken to mitigate or hide the effects of extremely long latency main memory accesses - out-of-order execution and memory access reordering, for example - though one of the oldest and most effective is hardware caching.

Caches, which are small and fast banks of memory placed close to the processor core, store recently/frequently used data so that it can be quickly accessed if it is used again (saving many main memory requests). By servicing memory requests close to the core, these caches reduce average memory access latency, power usage, and improve overall memory bandwidth, making them critical to good performance. Modern architectures generally have multiple layers of progressively slower/larger caches - a small per-core “L1” and “L2”, along with a large and shared Last Level Cache (“LLC”) being the most common organization [18].

Caches are only beneficial if the data that they store is re-used; i.e., if requests to the caches are hits (as opposed to misses, which then require a main memory access). The hit rate of a cache depends on (1) the memory access patterns of the running code, (2) the replacement policy which decides what data should be stored in the cache, and (3) the size of the cache itself. This paper will focus on the third dependency - namely, how to increase the effective capacity of the cache through cache compression.

Cache compression is a compelling idea because it allows for potentially significant increases in the effective capacity of a cache, without having to significantly increase the on-chip-size of the cache itself (which can already take 15-30% of overall chip area). Prior work [21, 1, 6] has also shown that data in the cache is often highly compressible. The compressibility numbers in such prior work are often overstated (due to selection bias or choosing non-representative sections of benchmarks to simulate) [7], but there usually is still notable compressibility in benchmarks. As Figure 1.1 shows, for example, 23% of lines on average (14% median) in SPEC2006 benchmarks are at least 2x compressible, with several benchmarks having 40% or more accesses which are 2x or 4x compressible.

Other prior work [24, 23] has shown that compressible cache lines can be efficiently stored with only small space overheads compared to an uncompressed cache (2-4%), and
that highly compressible benchmarks can gain significant performance improvements from compression-aware caching policies/architectures [3, 11, 23, 20].

While prior work has shown that compressed caches have potential, no work has yet quantified the extent of this potential. There are fundamental limits to how many cache hits are possible for a given caching architecture and benchmark, depending on how frequently data is re-used, and how much data is re-used; this limit, also known as the caching optimal, is an important indicator of headroom - i.e., how much remaining performance is possible to achieve. The uncompressed caching optimal is simple and efficient to compute (linear in the length of the trace), but the compressed caching optimal is relatively poorly understood, suspected to be NP-complete, and has only been studied in the context of software caches [5]. Knowledge of the compressed optimal enables many insights: comparing the theoretical efficiency of different caching architectures, understanding how much benefit compressed caches give over uncompressed caches, and understanding how much better replacement policies can get, amongst others.

In order to better understand the benefits of compressed caching, and how various compressed caching algorithms compare, this paper provides a novel formulation of the compressed optimal as an integer linear program. Specifically, we make the following contributions:

1. We provide a formulation of the compressed optimal which is easy to extend to various compressed caching architectures, and which is feasible to compute for representative samples of realistic benchmarks (250M-1B instructions in length).
2. We show that many compressed cache architectures provide nearly-identical optimal hit rates on SPEC2006 benchmarks (implying that simpler architectures with more limitations may be preferable to more complex architectures).

3. We show that greedy approximations of the compressed optimal, which simply run Belady’s algorithm on a compressed cache, are relatively accurate on our set of SPEC2006 benchmarks (within 2-3% of the true optimal hit rate).

4. We show that there is significant headroom between the previous state-of-the-art compressed cache replacement policy and the compressed optimal.

5. To lessen the gap between compressed replacement policies and the compressed optimal, we extend the Hawkeye [14] replacement policy into the compressed domain, achieving an average 5.30% absolute (13% relative) hitrate improvement over YACC+RRIP, or 7.13% absolute (18% relative) hitrate improvement over CAMP on SPEC2006 benchmarks.

The remainder of this paper is organized as follows: in Chapter 2, we explain the relevant prior work and important ideas in compressed caching; in Chapter 3, we discuss how to compute the compressed optimal for various cache architectures; in Chapter 4, we present a compressed caching policy which outperforms the prior state-of-the-art; and in Chapter 5 we conclude with possible future directions and acknowledgements.
Chapter 2

Background

To an external viewer, a compressed cache operates the same way as an uncompressed cache does: it stores a variable number of cache lines - fixed-size regions of recently-used memory (usually 64B in size) - in one of a fixed number of cache sets. The differences are only apparent inside each cache set - compressed caches must have extra mechanisms to compress, store, and manage compressed cache lines. We discuss each mechanism in kind.

2.1 Cache-Line Compression: CPACK+Z

A hardware cache-line compression algorithm is responsible for compressing incoming cache lines as tightly as possible, while minimizing overall compression/decompression latency and hardware footprint. The performance constraints on cache compression algorithms are asymmetric: since cache lines can be compressed in the background, only the decompression step is on the critical path. As such, most cache compression algorithms focus on low decompression latency in exchange for higher compression latency.

Since the compression algorithm for a compressed cache system is part of the cache hardware (and thus must be implementable directly in silicon), and decompression must be low latency, it’s overall complexity must be low compared to much more advanced/thorough software compression schemes. As a result, most hardware compression schemes focus on simple but effective heuristics for compression: compressing integers with low dynamic range [21], eliminating common bit patterns [1], and eliminating zeros. More advanced (but slower) algorithms use small dictionaries to store/look up frequent byte patterns and store them more efficiently [6].

Since this paper is investigating optimal hit rates/upper bounds, we opt to use CPACK+Z [6], a dictionary-based approach which consistently achieves the highest compressibility ratios on our benchmarks. CPACK focuses on compressing lines just past important compressibility thresholds - 2x compressible, 4x compressible, and so on - to maximize benefit to the compressed cache. In exchange for high performance, however, it suffers from a relatively long 9 cycle decompression latency; other compression policies, like Base-Delta-Immediate [21], only have a 1 cycle decompression latency. CPACK+Z is simply CPACK with zero-detection, where cache lines of all zeros can be efficiently stored as a single bit or flag.
2.2 Cache-Line Storage: A Simple Classification

Different cache architectures opt for different ways to store compressed lines; each generally must each make tradeoffs between overhead (as compared to an equivalent-sized uncompressed cache), effective capacity (how effectively a cache architecture can exploit compressibility), complexity, and latency. For the purposes of this paper, they can be classified based on their chosen compaction strategy, which is the actual mechanism that the cache architecture uses to track and store compressed lines. The compaction strategies themselves can be organized along two primary axes: the method through which they increase the number of tags, and data layout of the actual cache lines.

The lookup table which a cache uses to determine if a given way is in the cache (and if it is, where it is) is canonically known as the tag array. Each entry in the tag array is a tag. Compressed caches can often store many more potential cache lines than an equivalently-sized uncompressed cache, which necessitates increasing the capacity of the tag array. There are two primary schemes used:

- **Larger Tag Arrays**: The most obvious way to increase the number of available tags in the tag array is to just increase the total size of the tag array. This imposes no extra restrictions on what can be stored in the cache, but potentially comes at significant space overhead: just doubling the size of the tag array in a modern LLC cache increases the total cache area by 6-7% [23]. Larger increases in the tag array size (to better exploit highly compressible workloads) would incur even higher overheads, making this approach very space-expensive. Most early compressed cache architectures, like variable-sized compression [2] and IIC-C [13], take this approach.

- **Superblocks**: To avoid the high space overhead of large tag arrays, some compressed cache architectures (most notably DCC [24], SCC [22], and YACC [23]) keep the size of the tag array the same and track superblocks instead of cache lines directly. Each superblock generally represents a 4-cache-line-sized region of memory (256 bytes, assuming a 64 byte cache line), and each superblock entry can efficiently track all four cache lines in it’s memory region (since almost all of the high-order memory address bits are shared between all four cache lines). While this incurs virtually no overhead (< 1%) over a normal tag array, it adds the restriction that the superblock tag array can only address extra compressed lines if the compressed lines belong to the same superblock (i.e., the compressed lines must be spatially local). While this can be a severe restriction for benchmarks without much spatial locality, prior studies have shown that many cache lines in a given superblock are often co-resident in the cache at the same time, implying there is often enough spatial locality to make superblocks practically useful [24].

Cache architectures must also choose a specific data layout, which defines how compressed cache blocks are stored physically (and how they are mapped to from a tag array). There is a lot of engineering freedom in this domain (back-pointers vs. forward-pointers vs. data blocks), which affects latency and overhead, but from the perspective of a compressed replacement policy, there are two notable varieties:
• **Heterogeneous:** Heterogeneous architectures do not have a concept of a cache “way” (or they have a very permissive definition of a way), and allow as many cache lines to be stored in the cache as the cache capacity and tag array support. As a result, cache architectures do not have to worry about how to optimally pack cache lines into ways, though such freedom may incur high overheads and complex mechanisms like back-pointers or alignment networks. Heterogeneous caches are typically implemented by splitting incoming cache lines into small, fixed-size “sectors” (which are a size small enough to support all the possible compressed cache line sizes), and referencing these sectors via linked lists of pointers. The most notable heterogeneous architecture is DCC [24].

• **Homogeneous:** Homogeneous architectures retain the concept of a cache way from uncompressed caches (though they may call them other names, such as “data segments” or “data blocks”), and additionally enforce the constraint that all cache lines in a given way must have the same compressibility. These restrictions allow for simplistic tag array mapping strategies (like direct-mapping, which just directly associates a tag with a cache way), and incur comparatively low space/energy overheads, but either restrict how cache lines can be stored in the cache, or shift the burden of optimally grouping cache lines (in order to minimize internal fragmentation) to the replacement policy.

Other potential variants exist (such as a *way-heterogeneous* cache, which supports differently-sized cache lines in the same way), but virtually all published compressed cache architectures fall into one of the two aforementioned types.

**Yet Another Compressed Cache**

While this work will compute optimal hit rates for all of the compressed architecture types mentioned above, Chapter 4 (Compressed Hawkeye) will specifically use Yet Another Compressed Cache (YACC) [23]. YACC is a superblock-homogeneous compressed cache architecture with empirical performance similar to more complicated architectures, like Sectored Compressed Cache (SCC) [22] and Decoupled Compressed Cache (DCC) [24].

### 2.3 Cache-Line Replacement: YACC and CAMP

Cache-line replacement policies, generally just known as “replacement policies” or “eviction policies”, are responsible for deciding what lines should currently be in the cache at any given time. Virtually all replacement policies operate by always putting any newly accessed cache lines into the cache, and choose which line to evict based on various heuristics. Compressed replacement policies are no exception, though their task can be slightly more difficult: unlike uncompressed caches, where only one cache line needs be evicted to make space for an incoming cache line (since all lines are the same size), compressed caches may need/want to evict *multiple* compressed lines to make space for a single incoming line.

This paper will be referencing four replacement policies: two compressed, and two uncompressed, each of which is described below. Note that we consider YACC+RRIP to be the
state-of-the-art compressed cache replacement policy, and Hawkeye to be the state-of-the-art uncompressed policy.

**Least Recently Used (LRU)**

The Least Recently Used heuristic operates by, as the name implies, evicting the least recently used item whenever a new cache line arrives. It is one of the most popular cache replacement schemes in virtually all caching contexts, thanks to its strong baseline performance and simplicity of implementation. We use a simple uncompressed variable of it as a performance baseline when measuring Instructions Per Cycle (IPC).

**YACC + Re-reference Interval Prediction (RRIP)**

Re-reference Interval Prediction (RRIP) [15] works by associating each cache-line with a predicted re-reference time: a number between $0 - (2^N - 1)$, where 0 means the line is predicted to be re-used immediately, and $2^N - 1$ means the line is predicted to be used in the distant future. The cache line with the highest predicted re-reference time is evicted whenever an incoming cache line arrives. RRIP has two mechanisms for assigning predicted re-reference times: Static RRIP (SRRIP), which assigns every incoming line a value of $2^N - 2$, and Bimodal Interval Prediction (BIP), which randomly assigns some lines a value of 0 and some lines a value of $2^N - 2$. On a hit, the hit cache line’s time is set to 0 (since an access usually indicates another access in the future); on a miss, all cache lines predictions are incremented by one (since a miss indicates the cache lines are less likely to be used). This paper uses Dynamic RRIP, which dynamically chooses to use SRRIP and BIP at runtime depending on which is performing better; it runs on the YACC cache architecture at the superblock level.

**Compression-Aware Management Policy (CAMP)**

The Compression-Aware Management Policy (CAMP) [20] is a policy which builds upon RRIP, though it runs on a plain homogeneous cache architecture (i.e., it does not use superblocks), and it explicitly only uses SRRIP instead of DRRIP. CAMP has two primary insights: first, that compressed cache lines should preferentially be kept in the cache, and that the compressed size of a cache line can be a predictor of future re-use. To bias the cache towards keeping compressed lines, CAMP computes the “value” of a cache line as $\frac{\text{RRIP value}}{\text{compressed size}}$, and repeatedly evicts lines with the highest value until space is available for the incoming line. To take the size of the cache line into account, CAMP dynamically estimates whether cache lines of a certain size should be inserted at “high priority” (RRIP value $= 0$) or “low priority” (RRIP value $= 2^N - 1$).
Hawkeye

Hawkeye [14] is a state-of-the-art uncompressed caching policy built on top of SRRIP which runs Belady’s MIN [4] on past accesses to obtain optimal past caching decisions; these optimal caching decisions are used to train a program-counter based predictor, which then decides whether future accesses should be prioritized ("cache-friendly", RRIP value $= 0$), or de-prioritized ("cache-averse", RRIP value $= 2^N - 1$).
Chapter 3

The Compressed Optimal

One of the most direct ways to quantify the value of a caching architecture (including a compressed caching architecture) is to compute its optimal hitrate on various benchmarks: the absolute best hitrate that any policy could achieve using the given caching architecture on a specific benchmark, even if the policy had perfect knowledge of the future. Optimal hit rates are a direct reflection of both the memory access patterns of the benchmark and the restrictions of the caching architecture; by definition, they are also an upper bound on the hitrate that any replacement policy can achieve on the specific architecture/benchmark. They are useful for two primary reasons:

1. As a way to compare different cache architectures: Since optimal hit rates only depend on the benchmark and caching architecture, they allow for the maximum potential hitrate of different architectures on a set of benchmarks to be compared directly.

2. As a way to understand headroom: Optimal hit rates are usually infeasible for normal replacement policies to obtain, since “optimal” policies (such as Belady’s MIN) require knowledge of the future. However, the gap in performance between the optimal and a given replacement policy is a useful indicator of how much room for improvement there is: large gaps imply that there are still unexploited/underutilized memory access patterns that the replacement policy could use, while small gaps imply that not much more can be done to improve.

The remainder of this section will focus on the first use case, by (1) describing the uncompressed optimal and why it cannot simply be extended into the compressed domain, (2) providing an exact formulation of the compressed optimal via integer linear programming, (3) presenting optimal hitrate results for 3 different compressed cache architectures, and (4) using these results to understand how compressed caching theoretical compares to an uncompressed cache. The second use-case will be investigated in Chapter 4.

3.1 Belady’s Algorithm and Usage Intervals

The canonical algorithm for computing the optimal hitrate of an uncompressed cache is Belady’s MIN [4] algorithm, which requires knowledge of the future re-use time of every
Belady’s algorithm operates by evicting the line reused furthest in the future whenever a new line needs to be brought into the cache; if the line reused furthest in the future is the incoming line, then the incoming line is not cached. Cache lines which are never reused are assumed to have infinite time until their next reuse (so they are always evicted first). Belady’s algorithm is intuitively appealing: it chooses to cache the lines which are reused the earliest, so that cache space is available earlier for future caching decisions. The algorithm can be computed efficiently in two linear-time passes over a program execution trace: once to compute the re-use times of every access, and once to compute the actual caching decisions.

Belady’s algorithm is often visualized using usage intervals. A usage interval is the period of time from an access to a cache line to that cache line’s next reuse; for example, an access to cache line 0xDEADBEEF at time 7, with it’s next reuse at time 27, would form the usage interval (7, 27). A usage interval is the exact period of time for which the cache line must be resident in the cache in order for it’s next reuse to be a hit; if the cache line is evicted before it’s usage interval is complete, then the reuse is guaranteed to be a miss. Thus, the caching problem can be simplified to simply asking whether each usage interval should be cached in its entirety, or not at all. This simplification enables a more visually intuitive way to understand Belady’s algorithm: greedily cache the cache line with earliest reuse time that still fits in the cache. An example of Belady’s algorithm visualized using usage intervals on the access trace (ABCBDACCB) is shown in Figure 3.1.

3.2 Belady’s Algorithm for Compressed Caching

While Belady’s algorithm is simple to implement and fast to run, it is not optimal in the presence of variable-size cache lines, such as in compressed cache hierarchies. Consider the simplest possible example, with a size one cache and six accesses (or equivalently, three usage intervals) in Figure 3.2.

Belady’s algorithm fails to find the optimal decisions due to the critical difference between uncompressed and compressed caching architectures: in a compressed caching architecture, as Figure 3.2 shows, it is possible to obtain more overall hits by caching multiple overlapping cache lines in the cache.
Figure 3.2: A visualization of the access stream ABCABC, with a cache size of one way. The A interval is 1x compressible, and both the B and C intervals are 2x compressible. Belady would choose only the single bolded usage interval, since it only considers reuse time; the optimal would choose the two usage intervals shaded in gray.

Figure 3.3: A visualization of the access trace ABCBDACC, with A, C1, and B2 being 4x compressible. Belady’s algorithm will choose the two outlined usage intervals, while the true compressed optimal is the three shaded usage intervals.

compressible usage intervals in one way, even if the compressible usage intervals have greater reuse distances than uncompressible intervals in the same region. This is a general difference which goes beyond just the trivial example shown; another example is shown in Figure 3.3. It is important to emphasize, however, that it is also possible for caching multiple uncompressible usage intervals to still be preferable over caching multiple compressible intervals, as shown in Figure 3.4.

The above facts imply that both reuse distance and usage interval compressibility are important to the compressed optimal, and that one does not dominate the other. Since Belady’s algorithm does not take compressibility into consideration, it is insufficient in general for computing the compressed optimal. That being said, it is still an effective approximation for the compressed optimal in two scenarios:

1. Most lines are of similar compressibility: If most lines are similar in compressibility (such as if > 90% of lines are 2x compressible), then Belady’s algorithm can be made near-optimal, by assuming the cache has a larger overall capacity and ignoring the actual compressibility of the lines.
Figure 3.4: A visualization of the access trace ABCCCCAB for a size one cache, where the A
and B usage intervals are 2x compressible and the C usage intervals are 1x compressible.
Despite A and B being compressible, it is more optimal to cache all of the uncompressible
C usage intervals (for a total of three hits).

2. There is not much overlap between compressible lines: Belady’s algorithm is only sub-
onoptimal in the specific case when the policy must make a decision between caching
an uncompressible line, and caching multiple compressible lines which all overlap. If
there isn’t much overlap between different compressible lines, or not much overlap be-
 tween any uncompressible lines and any compressible lines, then Belady’s will also be
near-optimal since reuse time dominates the decision.

Obtaining accurate optimal hit rates for compressed caches when the two above scenarios
do not apply - i.e., for mixed-compressibility benchmarks with significant overlap between
compressible lines - is the focus of the next section.

3.3 Computing the Compressed Optimal

Due to there being a complex tradeoff between reuse distance and line compressibility when
making caching decisions for compressed caches, it is not clear if there is a greedy algorithm
for exactly computing the compressed optimal, or if there is a polynomial time algorithm at
all. In the absence of an efficient algorithm, the most direct way to compute the compressed
optimal on $n$ usage intervals is to try all possible cache/don’t cache combinations on the
intervals, which would have a $O(2^n)$ runtime. This runtime is intractable for anything more
than toy examples (with a practical limit of $n \approx 30 - 40$); it could heuristically be reduced
in common cases by only branching when the decision is ambiguous, but this would require
discovering and implementing numerous heuristics which depend on each specific caching
architecture.

A simpler way to make computing the compressed optimal tractable is to phrase it as an
integer linear programming problem, where each usage interval is represented by a boolean
variable $u_i$ (where $u_i = 0$ means the line would not be cached, and $u_i = 1$ means the line
would be cached), and the cache architecture restrictions are phrased as additive constraints
on these variables. The optimal hitrate would be the maximal value of $\sum_{i=1}^{n} u_i$ obtainable
with the given constraints. Integer linear programming is NP-complete, and in the worst
case will also degrade to a $O(2^n)$ exponential search over all assignments of $u_i$, but modern
ILP solvers can often find optimal or near-optimal solutions in reasonable time frames for
hundreds of thousands of variables. Practically speaking, using an ILP solver raises the tractable number of accesses from around 30, to around 2,500 (where “tractable” means computable on commodity hardware with an open source ILP solver in a few days); if we allow a 1% error in the final hitrate estimate, then the tractable number of accesses raises to 20,000. Using commercial solvers such as Gurobi [12] or CPLEX [8] could raise this limit higher.

Two aspects of virtually all caching architectures make the 20,000 access limit reasonable:

1. **Caches are set-associative**: Virtually all large caches are set associative, so solving the overall optimal hitrate problem is equivalent to solving many smaller optimal hitrate problems - one per set. For a modern 2MB LLC cache with 2048 sets, this means a 10,000,000 access stream effectively becomes two thousand 5,000-access problems, each of which is tractable.

2. **Only reused cache lines need to be considered**: Addresses/cache lines which are only accessed once (such as in streaming memory patterns) will never be cached by the optimal caching policy, since they are never reused. They can safely be assumed to be misses, and excluded from the optimal computation, reducing the total number of accesses.

The remainder of this section will present the ILP formulations of three different compressed cache architectures: heterogeneous, homogeneous, and superblock-homogeneous, as they are described in Chapter 2. The formulation for the superblock-heterogeneous caching architecture is a straightforward application of techniques we present in the other formulations, so we will not discuss it here.

3.3.1 Heterogeneous Optimal

The simplest cache architecture to express as an ILP problem is a *fully heterogeneous* compressed cache: i.e., a cache which can store any number of compressed lines so long as the sum of all of their sizes is less than the total cache capacity. Given a set of usage intervals, let \( u_i \) be the indicator variable for whether the \( i \)th usage interval would be cached, and \( s_i \) be the \( i \)th usage intervals corresponding size. Additionally, let \( t_{\text{MAX}} \) be the largest usage interval end time (i.e., \( t_{\text{MAX}} \) is the last time quanta for any usage intervals), let \( S \) be the total size/capacity of the cache, and let \( U_t \) be the set of the indexes of all usage intervals which cross time \( t \). With all of these variables, we can succinctly represent the ILP for a heterogeneous cache:

\[
\begin{align*}
\text{Maximize} & \quad \sum_{i=1}^{n} u_i \\
\text{Subject to} & \quad \forall i \in [0, n - 1], \quad u_i \in \{0, 1\} \\
& \quad \forall t \in [0, t_{\text{MAX}}], \quad \sum_{i \in U_t} s_i \cdot u_i \leq S
\end{align*}
\]
This formulation of the heterogeneous optimal is effectively a literal translation of the restrictions of a heterogeneous cache: the first restriction on $u_i$ forces all cache lines to either be cached/not cached, and the second restriction reinforces that, at every time quanta, the total size of the lines in the cache is less than the cache capacity. For $n$ usage intervals, there are exactly $n$ variables, and $n + t_{MAX}$ constraints.

3.3.2 Homogeneous Optimal

The optimal hitrate for a homogeneous cache is slightly more involved to represent as an ILP, due to the fact that all lines in a given cache way must have the same compressibility. One way to enforce this constraint is to use representative variables: for each distinct compressibility level $c$ and time $t$, create a “representative” $w_{t,c}$, which represents the number of cache ways of the given compressibility that are needed to store all of the cached usage intervals with compressibility $c$. In other words,

$$w_{t,c} = \left\lceil \frac{\sum_{i \in U_{t,c}} u_i}{c} \right\rceil$$

where $U_{t,c}$ is the set of indexes of usage intervals which contain time $t$ and have compressibility $c$. The formula as it exists above is not directly convertible to ILP (as the ceiling function is non-linear), but we can emulate it with the equivalent representation

(1) $w_{t,c} \in \mathbb{N}$

(2) $\sum_{i \in U_{t,c}} u_i \leq c \cdot w_{t,c} \leq (c - 1) + \sum_{i \in U_{t,c}} u_i$

These representative variables implicitly enforce the restriction that a given cache way must have lines of the same compressibility; enforcing the overall cache capacity constraint is then as simple as ensuring that the total number of ways used of each compressibility $c \in C$ is less than the cache capacity $S$ at every time $t$:

$$\sum_{c \in C} w_{t,c} \leq S$$

Putting everything together yields the final ILP formulation:

Maximize $\sum_{i=1}^{n} u_i$

Subject to $\forall i \in [0, n-1], \ u_i \in \{0, 1\}$

$\forall t \in [0, t_{MAX}], c \in C, \ w_{t,c} \in \mathbb{N}$

$\forall t \in [0, t_{MAX}], c \in C, \ \sum_{i \in U_{t,c}} u_i \leq c \cdot w_{t,c} \leq (c - 1) + \sum_{i \in U_{t,c}} u_i$

$\forall t \in [0, t_{MAX}], \sum_{c \in C} w_{t,c} \leq S$

(3.2)
The introduction of the representative variables notably increases the number of variables to \( n + |C| \cdot t_{MAX} \), and the number of constraints to \( n + 2|C| \cdot t_{MAX} + t_{MAX} \). In practice, the possible set of compressibilities \( C \) is a small set of powers of two (such as \( C = \{1, 2, 4\} \)), so the resulting number of variables is still tractable. There are also methods to significantly reduce the number of generated representative variables/constraints by lazily generating them only when they are needed - these methods are described in the following section on the Superblock-Homogeneous Optimal.

### 3.3.3 Superblock-Homogeneous Optimal

The superblock-homogeneous optimal is similar to the plain homogeneous optimal, except we now need to create representative variables for potentially every \((\text{time}, \text{compressibility}, \text{superblock})\) triple, which can quickly lead to a quadratic blowup in the total number of variables and make the problem infeasible even for relatively small numbers of usage intervals. Fortunately, it is possible to do better: if a usage interval at a given time does not overlap any other usage intervals with the same compressibility and superblock (or if the usage interval is just uncompressible), then there is no point in making a representative variable - the usage interval will either be cached by itself in one way at that time, or it won’t be cached at all (so just using \( u_i \) itself as the representative variable is sufficient).

With this optimization in mind, the formulation needs to add an extra pre-processing step before defining the ILP. First, construct a map \( O_t[c, s] \), which maps \((\text{compressibility}, \text{superblock})\) pairs to the set of intervals which cross time \( t \), and which have the corresponding compressibility/superblock tags. We then use this map to construct the set of representatives \( R_t \) at each time \( t \); the sum of this set of representatives is the total number of cache lines used by the currently cached lines. For each unique \((c, s)\) pair in \( O_t[c, s] \):

- If \( c = 1 \), then the usage interval can’t be co-located with any other usage intervals regardless of how many there are, because it is uncompressible. Thus, we can use the interval indicator variable itself as the representative. I.e., \( R_t = R_t \cup O_t[c, s] \).

- If \( \text{len}(O_t[c, s]) = 1 \) (i.e., \( O_t[c, s] = \{ u_i \} \) for some \( i \)), then there are no other intervals which this lone interval could be co-located with in the same way at this time (even if the line is compressible), so we can use the interval indicator variable itself as the representative. I.e., \( R_t = R_t \cup \{ u_i \} \).

- Otherwise, \( \text{len}(O_t[c, s]) > 1 \) and there are multiple usage intervals with the same compressibility/superblock at the given time, which might be co-located with each other. This is the only instance where we need to introduce a representative variable \( w_{t,c,s} \), defined similarly as in the homogeneous case:

\[
\begin{align*}
(1) w_{t,c,s} & \in \mathbb{N} \\
(2) \sum_{u \in O_t[c,s]} u & \leq c \cdot w_{t,c,s} \leq (c - 1) + \sum_{u \in O_t[c,s]} u
\end{align*}
\]

Only the representative variable is added to the set of representatives; i.e., \( R_t = R_t \cup \{ w_{t,c,s} \} \).
After this preprocessing step, the ILP formulation is straightforward. For convenience, define $R_t^*$ to be the specially introduced representative variables at time $t$; i.e., $R_t^*$ contains all of the created $w_{t,c,s}$ variables at time $t$. Then,

Maximize $\sum_{i=1}^{n} u_i$

Subject to

$\forall i \in [0, n-1], u_i \in \{0, 1\}$
$\forall t \in [0, t_{MAX}], w_{t,c,s} \in R_t^*, r \in \mathbb{N}$
$\forall t \in [0, t_{MAX}], w_{t,c,s} \in R_t^*, \sum_{u \in O_t[c,s]} u \leq c \cdot w_{t,c,s} \leq (c-1) + \sum_{u \in O_t[c,s]} u$
$\forall t \in [0, t_{MAX}], \sum_{r \in R_t} r \leq S$

While this method does not improve the worst-case number of variables for a given set of intervals, it dramatically reduces the number of extra representative variables (and corresponding constraints) in practice; the number of generated representative variables was observed to generally be $O(n)$, rather than the worst case $O(n^2)$. It is worth noting that this optimization can also be applied to the plain homogeneous case, by simply setting $s = 0$ everywhere (or omitting it altogether).

### 3.3.4 Other Methods

There are other ways to compute or approximate the compressed optimal; we discuss them here briefly.

**Exact Methods**

There are two other notable methods to exactly compute the compressed optimal which we considered but ultimately did not pursue: PFOO [5], which phrases compressed caching as a min-flow max-cost graph problem; and CSOPT [16], which phrases cost-sensitive caching as a search tree problem with pruning (and which can be adapted to compression-aware caching). Both approaches are theoretically viable and could work in the compressed caching domain; however, we had practical problems implementing or using them which motivated our Integer Linear Programming formulation. In particular, PFOO’s source code was difficult to work with and slow to run; our initial results were effectively nonsensical, and our debugging proved fruitless so we moved on to other approaches.

The approach taken by the CSOPT paper is more general, since it views compressed caching as a “game” (as in game theory), where the decision points are what line to evict at each cache miss. Specifically, to compute the optimal, the paper does an exhaustive search through the game tree of all possible caching decisions; to make this tractable to compute, it applies several general “tree pruning” rules which allow it to discard/ignore vast parts of the search tree. This method could very well also work for the compressed optimal, and have good practical performance; we ultimately did not pursue it because coming up with
good pruning rules is potentially time intensive and these pruning rules may not generalize between different caching architectures.

Greedy/Approximate Methods

While Section 3.2 shows that Belady’s algorithm is suboptimal in the general case, it also notes that the algorithm can often still be a good approximation for the optimal in cases with very low or very high amounts of compressibility, or where there is not much overlap between compressible lines and uncompressible lines. This intuition can become the basis for a greedy approximation to the compressed optimal, whereby we simply run Belady’s algorithm on a model of a compressed cache architecture. Since we model the underlying compressed cache architecture, the algorithm can still incidentally take advantage of compressibility (by unknowingly caching a compressed line, which allows it to then cache another compressed line). This approximation can be implemented in a constant amount of space and - perhaps more importantly - can be computed incrementally in a single linear time pass through the trace; in practice, it is a simple extension to the OPTgen vector from Hawkeye [14].

3.4 Empirical Results on Benchmarks

We now present the optimal hit rates in the Last Level Cache (LLC) for the three caching architectures (Heterogeneous, Homogeneous, and Superblock-Homogeneous) described above, on 22 memory-intensive SPEC 2006 benchmarks. Benchmark access traces were 250M instructions long and were generated using SimPoints [25] and the Intel PIN toolset [19]. Conversion from the full access trace to a textual LLC access trace was done by the ChampSim CRC2 out-of-order cache simulator [17], using the specifications detailed in Figure 3.5. Finally, the problem was converted to an ILP and solved using cvxpy [9] and the CBC ILP solver [10]. Compressibility information was obtained by the CPACK+Z compression algorithm; only 1x, 2x, and 4x compressible lines were considered in the computation. Figure 3.6 contains the per-benchmark optimal hit rates, while Figure 3.7 is the aggregate optimal hit rates.

The most critical takeaway from these results is that all three of the cache architectures have very similar optimal hit rates. The most restrictive architecture - superblock-homogeneous, which has restrictions on both the tag array and the data layout - has an average hitrate across all benchmarks only 0.9% less than the most permissive architecture.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Single core, out-of-order</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td>32KB set-associative (64 sets, 8 ways), 4 cycle latency</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256KB set-associative (512 sets, 8 ways), 8 cycle latency</td>
</tr>
<tr>
<td>LLC Cache</td>
<td>YACC; 2MB set-associative (2048 sets, 16 ways), 20 cycle latency (+9 cycles if accessing compressed line)</td>
</tr>
<tr>
<td>Main Memory</td>
<td>200 cycle latency</td>
</tr>
<tr>
<td>Compression</td>
<td>CPACK+Z compression algorithm; 1x/2x/4x only.</td>
</tr>
</tbody>
</table>

Figure 3.5: Cache simulation parameters used when obtaining results.
Figure 3.6: Comparison of three compressed caching optimals; bolded entries are “compressible” benchmarks, which we define as benchmarks with an average line compressibility of 1.4 or higher.

Figure 3.7: Average hitrate across all benchmarks, and across “compressible benchmarks” (average line compressibility of 1.4 or greater), for three compressed optimals.
The homogeneous architecture has nearly identical performance to the heterogeneous architecture on all benchmarks, and has an average hitrate that is only 0.09% lower than the heterogeneous average hitrate (0.12% on compressible benchmarks). There are several probable reasons for this apparent lack of difference:

- **Architecture differences are not relevant in practice**: The most literal reasoning for why the cache architectures all perform similarly is that the mechanisms that set them apart were not actually exercised in the test benchmarks. An architecture can only exploit as much potential as is already present in the data trace: if the amount of extra hit rate potential for compressed lines is low, then all of the architectures will likely achieve that low potential and be unable to differentiate themselves from the other architectures. This reasoning is backed up by the fact that the difference between the uncompressed optimal and the compressed optimals, as discussed in the following sections, is relatively low.

- **Superblocks are relatively effective**: While there is some opportunity cost to using superblocks (based on the 0.9%/1.7% lower average hitrate on all benchmarks/compressible benchmarks), the opportunity cost is quite small, which implies that superblocks are an effective mechanism for efficiently creating extra tags in the cache.

- **Compressibilities tend to be powers of two**: In hardware compressed caches, compressed cache lines are usually “rounded down” to the nearest power of two compressibility (1x, 2x, 4x, etc.) for overhead/efficiency reasons. Having power-of-two compressibilities significantly reduces the chance of internal fragmentation of data within each way in a homogeneous architecture as compared to a heterogeneous architecture, reducing the benefits of a purely heterogeneous architecture. For an example, consider if every incoming cache line could be compressed from 64 to 33 bytes: the homogeneous architecture would suffer significant fragmentation, since it could only put one compressed line per way, while the heterogeneous architecture would get almost double the effective capacity.

### Accuracy of Greedy Approximations

We now present a comparison between greedy approximations of compressed caching optimals (which amount to just running Belady’s on different cache models, as described in Section 3.3.4), and the actual true optimal values. The per-benchmark comparisons can be found in Figure 3.8 (greedy superblock-homogeneous), and Figure 3.10 (greedy heterogeneous); the corresponding overall average comparisons can be found in Figure 3.9 and Figure 3.11 respectively.

The greedy approximations are clearly suboptimal, though they are surprisingly close to the actual true values - the greedy version of superblock-homogeneous optimal only has an average error of 1.78% (1.69% on compressible benchmarks); and the greedy version of the heterogeneous optimal has an average error of 1.82% (1.83% on compressible benchmarks). As they can be computed in linear time and linear space (or further approximated in constant space), they may be reasonable alternatives to expensive ILP computations for very long
Figure 3.8: A per-benchmark comparison between the greedy computation of the superblock-homogeneous optimal, and the actual superblock-homogeneous optimal. Notably compressible benchmarks (average line compressibility of 1.4 or higher) have been bolded.

Figure 3.9: Average hitrate across all benchmarks, and across “compressible benchmarks” (average line compressibility of 1.4 or greater), for the greedy superblock-homogeneous optimal and the actual superblock-homogeneous optimal.
Figure 3.10: A per-benchmark comparison between the greedy computation of the heterogeneous optimal, and the actual heterogeneous optimal. Notably compressible benchmarks (average line compressibility of 1.4 or higher) have been bolded.

Figure 3.11: Average hitrate across all benchmarks, and across “compressible benchmarks” (average line compressibility of 1.4 or greater), for the greedy approximation of the heterogeneous optimal, and the actual heterogeneous optimal.
traces, or useful to get an initial estimate at how much a benchmark will benefit from compressibility.

**Headroom over Uncompressed Caches**

Next, we compare the uncompressed optimal with the superblock-homogenous optimal, to get a (slightly conservative) sense of how much theoretical benefit compression gives. Per-benchmark results are shown in Figure 3.12; the average hitrates are shown in Figure 3.13.

The most important takeaway from this comparison is that the overall potential hit rate improvement from compressed caching is relatively small: compression increases optimal hit rates by just 2.81% on average across all benchmarks, and just 3.44% on average for compressible benchmarks. Very small hitrate increases for low-compressibility benchmarks like 459.GemsFDTD or 462.libquantum are understandable: compressed caching cannot benefit benchmarks which don’t have compressed cache lines. However, many highly compressible benchmarks suffer from small hit rate increases as well: 453.povray, despite 40% of cache lines being 2x compressible, obtains no hit rate improvement at all. Similarly, 465.tonto achieves only a 2% hitrate improvement despite also being over 40% 2x compressible. It is worth noting that some compressible benchmarks do achieve large hit rate increases - for example, 436.cactusADM has an 10% hit rate improvement.

The crux of the issue is that compression is only useful to benchmarks which are both compressible and cache-friendly; many of the benchmarks in SPEC2006 seem to be oblivious to the size of the cache (such as 453.povray) or have low compressibility (such as 462.libquantum), which makes compressed caching yield overall low hit rate improvements. It is unclear whether this scenario is specific to the SPEC 2006 benchmarks, or if it is a more general phenomenon.

**Headroom in Existing Replacement Policies**

Finally, we compare the compressed optimal to two prior state-of-the-art compressed replacement policies: YACC+RRIP, and CAMP. Policy performance numbers were collected using the ChampSim CRC2 out-of-order cache simulator [17], using the same traces that were used in the previous optimal computations. Per-benchmark comparisons can be found in Figure 3.14; overall average comparisons can be found in Figure 3.15.

These results show that there is significant headroom (12% absolute hitrate improvement, or 27% relative hitrate improvement) between prior state of the art policies and the compressed optimal; while reaching the performance of an optimal is generally infeasible, such large gaps imply that it is definitely possible to do better. There is an important caveat to this statement, however - since the uncompressed optimal and compressed optimal are relatively close in our set of benchmarks (3%), this also implies that there is a large gap (8-9%) between these policies and the uncompressed optimal. In other words, a significant amount of the available headroom for these policies can be obtained by simply doing a better job exploiting access patterns in general, independent of compressibility. We explore this idea further in Chapter 4.
Figure 3.12: A comparison between Belady’s algorithm (i.e., the uncompressed optimal), and the superblock-homogeneous compressed optimal. Notably compressible benchmarks (average line compressibility of 1.4 or higher) have been bolded.

Figure 3.13: Average hitrate across all benchmarks, and across “compressible benchmarks” (average line compressibility of 1.4 or greater), for the uncompressed optimal and the superblock-homogeneous compressed optimal.
Figure 3.14: A per-benchmark comparison between two state-of-the-art compressed replacement policies, and the actual homogeneous optimal.

Figure 3.15: Average hitrate across all benchmarks, and across “compressible benchmarks” for two state-of-the-art compressed replacement policies, and the actual homogeneous optimal.
Chapter 4

Compressed Hawkeye

As Section 3.4 shows, there is a significant performance gap (12%) between state of the art compressed cache replacement policies and the compressed optimal. Only 3% of this gap can actually be explained by compressibility, at most: the remaining 9% is due to compressed replacement policies failing to take advantage of potential access patterns. It thus makes natural sense to look at the much more mature field of uncompressed replacement policies, which focus on exploiting general access patterns, for inspiration. Doing so reveals something quite interesting: state-of-the-art uncompressed caching policies have better performance than current state-of-the-art compressed caching policies, even though they run on uncompressed caches! Figures 4.1 and 4.2 compare two compressed replacement policies, CAMP and YACC+RRIP, with Hawkeye, a state-of-the-art uncompressed replacement policy. Hawkeye outperforms the two compressed replacement policies on a majority of the benchmarks, sometimes by significant margins (see, for example, 456.hmmer and 459.GemsFDTD).

The fact that Hawkeye performs so well on an uncompressed cache, relative to policies running on a compressed cache, motivates our final contribution in this paper: to create a good compressed replacement policy, we can just modify a good uncompressed replacement policy. We dub the resulting replacement policy “Compressed Hawkeye” (very inventively). As Section 3.4 shows that there is not a significant opportunity cost to running on more restrictive cache architectures, Compressed Hawkeye runs on the YACC superblock-homogenous architecture by default (though it will also run on other architectures). Two notable changes are required to Hawkeye to make it perform well on a compressed cache:

1. Compression-aware eviction policy: As noted in Chapter 2, compressed replacement policies must be able to handle evicting several lines to make space for a single incoming line. This problem is worse in homogenous or superblock-homogenous caches: there may technically be free space in the cache for the incoming line which cannot be used because the tag array cannot track the incoming line. Compressed Hawkeye solves this using a score based heuristic similar to CAMP: for every way or subblock that the incoming line could be placed, the sum of the “value” of the blocks that would need to be evicted is computed. The place with the highest overall value is evicted. By default, Compressed Hawkeye simply uses the RRIP value of each cache line as it’s value - factoring in the compressed size of the cache line worsened performance.
Figure 4.1: A per-benchmark hit rate comparison between the compressed state-of-the-art replacement policies - CAMP and YACC+RRIP - and a state-of-the-art uncompressed replacement policy, Hawkeye.

Figure 4.2: Average hitrate across all benchmarks, and across “compressible benchmarks” for CAMP, YACC+RRIP, and Hawkeye.
2. **Compression-aware oracle:** Hawkeye’s core insight is to learn from optimal access decisions on the past. This is possible in uncompressed caches, since Belady’s algorithm can be computed in linear time and constant space via a simple-array based data structure. However, it is not possible to compute exact optimal access decisions for a compressed cache with the same time/space restrictions, so we instead choose to learn from a greedy approximation to the compressed optimal.

### 4.1 Empirical Results

We now evaluate Compressed Hawkeye by comparing it against CAMP and YACC+RRIP in terms of cache hit rate, and Instructions Per Cycle (IPC) improvement normalized to uncompressed LRU. All data was collected using the ChampSim simulator; relevant information about the simulation parameters can be found in Section 3.4, in Figure 3.5. The hitrates of the replacement policies are compared in Figure 4.3 and Figure 4.4. The IPC improvements of the policies (normalized to LRU) are compared in Figure 4.5 and Figure 4.6.

The hitrate improvements are dramatic: Compressed Hawkeye increases the absolute hit rate by 7.14% on average compared to CAMP (18% relative increase; 13% relative increase on compressible benchmarks), and by 5.30% on average compared to YACC+RRIP (13% relative increase; 10% relative increase on compressible benchmarks). Furthermore, Compressed Hawkeye notably lessens the gap between compressed replacement policies and the compressed optimal, from 12% to 7%. Even when factoring in the extra 9 cycle decompression latency (from CPACK+Z), these hitrate improvements translate to high (5%-30%) IPC improvements on cache-sensitive benchmarks compared to uncompressed LRU, and do not cause any notable performance degradation on cache-oblivious benchmarks, as Figure 4.5 shows. Compressed Hawkeye improves IPC by 4.11% on average compared to uncompressed LRU across all benchmarks; this translates to an average IPC improvement of 3.24% over CAMP and 2.30% over YACC+RRIP. Compressed Hawkeye also appears to capture most of the (admittedly limited) potential from compression for these benchmarks: the compressed version of Hawkeye achieves a 2.09% higher hitrate than the uncompressed version, which is around 75% of the full 2.81% possible average hitrate improvement from compression.

These results come from benchmarks with low or medium compressibility, like 482.sphinx3 (1.29 average cache line compressibility), 429.mcf (1.21), and 401.bzip2 (1.00). This appears to be a mixture of two factors: (1) Many compressible benchmarks in SPEC 2006 don’t benefit IPC-wise from better replacement policies or compressibility (403.gcc and 453.povray, for example); and (2) Hawkeye does a better job of exploiting access patterns in uncompressible and low-compressible benchmarks (as evidenced by Figure 4.1). Another interesting observation is that the long, 9 cycle decompression latency did not have a serious effect on any of the compressed policies: the average IPC was only 0.2% higher on average for all three compressed policies if the decompression latency was removed. This implies that even longer decompression latencies may be an acceptable tradeoff in exchange for more compressibility.
Figure 4.3: A per-benchmark comparison between the hitrates for CAMP, RRIP, and Compressed Hawkeye. Notably compressible benchmarks (average line compressibility of 1.4 or higher) have been bolded.

Figure 4.4: Average hitrate across all benchmarks, and across “compressible benchmarks” (average line compressibility of 1.4 or greater), for CAMP, YACC+RRIP, and Compressed Hawkeye.
Figure 4.5: A per-benchmark comparison between the IPC improvements (normalized to LRU) for CAMP, RRIP, and Compressed Hawkeye. Notably compressible benchmarks (average line compressibility of 1.4 or higher) have been bolded.

Figure 4.6: Average IPC improvement (normalized to LRU) across all benchmarks, and across “compressible benchmarks” (average line compressibility of 1.4 or greater), for CAMP, YACC+RRIP, and Compressed Hawkeye.
Chapter 5

Conclusion & Acknowledgements

5.1 Future Work

This project naturally motivates a large amount of potential future work; we list a few areas below.

1. More benchmarks: The SPEC2006 benchmark suite has a reasonable spread of different memory access patterns, but it tends strongly towards lower compressibilities. Investigating the compressibility and compressed optimals of more benchmarks - particularly server and mobile workloads - would be valuable in better understanding how much benefit compressed caching can give.

2. Understanding how performance scales with compressibility: Another worthwhile avenue for determining how valuable compression is is via artificial distributions: artificially increase/decrease the compressibility in benchmarks and see how it affects IPC/performance. This could also be used to set goals for how much compressibility a given compression algorithm would need to achieve for a given performance threshold.

3. Faster/more accurate compressed optimal approximations: There is still a lot of work that can be done in better approximating (or even exactly computing) the compressed optimal. More rigorously investigating other approaches - particularly the search tree based method - could yield much faster methods. Variants of Belady’s algorithm which incorporate some kind of lookahead, instead of being purely greedy, could also be fruitful.

5.2 Conclusion

In this thesis, we investigate compressed caching on SPEC 2006 benchmarks via the compressed optimal, and via Compressed Hawkeye, an extension of the Hawkeye replacement policy to the YACC compressed caches. Our results give mixed signals: compression can give significant hit rate and IPC improvement for some benchmarks (such as in 436.cactusADM, which achieves 11% higher hit rate/5% higher IPC), but it can also frequently accomplish
nothing (403.gcc, 444.namd, 447.dealII, etc.). In general, to make compression worthwhile in caching, we need to see consistently higher compressibilities, and significantly more cache-sensitivity in benchmarks. Prior work has claimed to see these higher compressibility numbers [21, 1, 6, 24, 23], but they are often exaggerated due to being measured at the L1 (instead of the LLC), the usage of unrealistically high compression ratios (8x/16x/32x), or selection bias when choosing benchmarks.

Furthermore, compressed replacement policies could also take more cues from uncompressed replacement policies: the Hawkeye replacement policy, which runs on an uncompressed cache, is notably more performant than several prior state-of-the-art compressed replacement policies, like CAMP and YACC+RRIP. This curious performance disparity may not be as prevalent on much more compressible benchmarks, but simplistic compressed replacement policies leave many potential hits on the table because they focus heavily on compression, rather than on access patterns.

More investigation is required to determine the true worth of compression in hardware caches, particularly focusing on more modern benchmarks like server or mobile workloads. Computing the compressed optimal and comparing it to the uncompressed optimal, or extending uncompressed replacement policies to run on compressed caches, could both be useful tools for understanding the value of compression in the future.

5.3 Acknowledgements

The author would like to thank Calvin Lin (advisor) and Akanksha Jain (honorary advisor) for their guidance, advice, and seemingly endless patience; Jack Youstra for his significant contributions towards making this thesis possible; and Jo Bridgwater for her commentary and revisions. Additionally, the author would like to thank the UT Computer Architecture reading group for their feedback on his thesis defense presentation.
Bibliography


