Lossy Compression for Worst-Case Execution Time Analysis of PLRU Caches

David Griffin
University of York, UK
david.griffin@york.ac.uk

Benjamin Lesage
University of York, UK
benjamin.lesage@york.ac.uk

Alan Burns
University of York, UK
alan.burns@york.ac.uk

Robert I. Davis
University of York, UK
rob.davis@york.ac.uk

ABSTRACT

This paper outlines how Lossy Compression, a branch of Information Theory relating to the compact representation of data while retaining important information, can be applied to the Worst Case Execution Time analysis problem. In particular, we show that by applying lossy compression to the data structures involved in the collecting semantics of a given component, for example a PLRU cache, a useful analysis can be derived. While such an analysis could be found via other means, the application of Lossy Compression provides a formal method and eases the process of discovering the analysis. Further, as the compression and its application are formally specified, such an analysis can be made correct-by-construction rather than relying on an after-the-fact proof.

1. INTRODUCTION

Real-time systems [7] are characterised not only by the need for functional correctness, but also the need for temporal or timing correctness. Real-time systems continually monitor and respond to stimuli from the environment and the physical systems that they control. In order for such systems to behave correctly, they must not only execute the correct computations, but also do so within predefined time constraints. These time constraints are typically expressed in terms of end-to-end deadlines on the elapsed time between a stimuli and the corresponding response. Applications in real-time systems may be classified as hard real-time, where failure to meet a deadline constitutes a failure of the application; or soft real-time, where latency in excess of the deadline leads only to a degraded quality of service. Today, hard real-time systems are found in many diverse application areas including; automotive electronics, avionics, space systems, medical systems, household automation, and robotics.

Worst Case Execution Time Analysis is one of the major areas of research in the real-time systems field. The Worst Case Execution Time (WCET) of a task is defined as the maximum amount of time which that task may require to execute. Worst Case Execution Time analysis is the problem of placing a bound on the execution time of a task. The term *tightness* is used to describe how accurate this upper bound is; a WCET estimate becomes tighter as it approaches the real WCET. Further, we use the term *soundness* to indicate that a WCET bound is indeed an upper bound that is not exceeded by the actual WCET [1].

One approach to the WCET problem is *Static Analysis*, which is the general term used to describe any WCET estimation technique which uses a model based on program source or assembly code and the behaviour of the hardware that it is to run on. A major difficulty with Static Analysis is State Explosion, where the number of states to check in the analysis grows exponentially. To combat state explosion, the two main techniques used in current approaches are Abstract Interpretation [11] and Symbolic Model Checking [6]. These techniques have the same goal of discarding information that is irrelevant to the problem at hand, but accomplish this goal using distinctly different methods. This paper focuses on Abstract Interpretation, although the ideas introduced are equally applicable to Symbolic Model Checking.

Collecting Semantics [12] is a specific technique implemented by Abstract Interpretation. Collecting Semantics seeks to utilise the fact that multiple concrete states may exhibit exactly the same behaviour. Hence, Collecting Semantics can provide a significant reduction in the number of states required for analysis, although this may not be sufficient to render the analysis tractable if a system contains a large number of distinct behaviours.

One problem in Abstract Interpretation is finding an appropriate approximation in order to discard irrelevant information. Abstract Interpretation relies on identifying a property of the system which can be usefully approximated [11], which leads to the situation where research is primarily focused on identifying such properties. Further, even when such a property is found, there is no guarantee that it will yield a useful approximation. A more fruitful approach would be to have a set of rules which give guidance, such that the innovative step can be designed to have the desired properties.

Information Theory [25] is the branch of mathematics dedicated to the analysis, handling and storage of Information. The original application of Information Theory was developed by Shannon in 1948 for use in analogue signal processing [23]. Specifically, analogue signals are prone to interference. Using Information Theory it is possible to specify precisely how much interference a signal can tolerate before

[1] We note that in the literature, this is sometimes referred to as safety or a safe bound; however, since safety has different connotations in safety-critical systems, we prefer the term soundness.

RTNS 2014 , October 8 - 10 2014, Versailles, France

http://dx.doi.org/10.1145/2659787.2659807.

it is unusable. Using similar ideas, Lossy Compression [26] is a technique that enables data of lower value to be discarded while preserving enough data such that the information is still useful for its intended purpose.

It can be argued that Abstract Interpretation implements a form of Lossy Compression, as Abstract Interpretation specifically sets out to discard data that is deemed by the implementer to be of low value. However, the technique of Abstract Interpretation provides no specific guidance on how one might determine what data is of low value. By contrast, Lossy Compression provides both a vocabulary and a method to determine low value data with regard to specific goals. This paper outlines how the techniques of Lossy Compression can be applied to Abstract Interpretation, with the aim of aiding the development of suitable abstractions.

1.1 Organisation

The remainder of the paper is organised as follows: Section 2 begins by describing the concepts of Abstract Interpretation and Collecting Semantics. A specific example of this is given for the PLRU Cache, using Grund and Reineke's PLRU cache analysis [17]. Section 3 gives an overview of Lossy Compression, in particular its application to MP3 [5]. This is combined with observations made about abstract interpretation to give an overview of how one may apply the principles of lossy compression to WCET analysis. PLRU cache analysis is revisited in Section 4 which describes how one can apply the principles of Lossy Compression to the PLRU cache, resulting in Full Tree Analysis, outlined in Section 5, a PLRU cache analysis method which does not discard important information. Full Tree analysis is evaluated against current methods in Section 6. Finally, conclusions about the application of lossy compression to static analysis and PLRU cache analysis in particular are given in Section 7.

2. ABSTRACT INTERPRETATION AND COLLECTING SEMANTICS

Abstract interpretation is a general program analysis technique developed by Cousot and Cousot [11]. The goal of abstract interpretation is to perform the minimal amount of work in order to prove the correctness of a property of a program. The method for this creates an abstract representation of the program which contains less information than the original program, but retains enough to provide a valid answer. For example, given the operation $-x \times y = -xy$, and the requirement that only the sign of the result was required, an appropriate abstraction would be $-(sign(x)) \times +(sign(y)) = -(sign(x)sign(y))$, completely discarding the magnitude of x and y and only using their signs. Then, once the abstract representation has been created, the abstraction can represent sets of concrete values with a single abstract value, without impacting the validity of the model. By also defining a merge function which combines sufficiently similar abstract points, the size of the model is reduced and hence a desired property becomes easier to prove. The level of approximation that the abstraction uses is not defined by Cousot and Cousot; instead it is left to the implementation and specific use to pick an appropriate level.

Collecting Semantics [12] are a form of Abstract Interpretation that seeks to preserve the operational behaviour, rather than concrete states. A simple example of Collecting Semantics is the approximation of the real numbers between $[0, 1]$ by the integers $0, 1$ using the floor function. As the floor function maps every real number to an integer, the behaviour of all real numbers is accurately represented by an integer. With the given ranges, the integer 0 represents the behaviour of all reals in the range $[0, 1)$, as $\lfloor x \rfloor = 0 = \lfloor 0 \rfloor$ for all $x \in [0, 1)$, with the integer 1 representing the behaviour of the real number 1. Hence applying the collecting semantics in this case results in a very large reduction in the number of states that need to be considered. In the case of the PLRU cache, a simple implementation of the collecting semantics is the *cache naming* approach, which is introduced in the next section.

2.1 Current Analysis of PLRU Caches

The Pseudo Least Recently Used (PLRU) cache is an approximation of an LRU cache. In empirical testing, the PLRU cache scheme achieves a hit ratio almost as good as an equivalently sized LRU cache [18]. However, as the PLRU cache structure is designed to easily map onto a silicon implementation, then for high associativities, the die area consumed by cache logic is greatly reduced compared to that required for an equivalent LRU cache [18]. This in turn results in the cache requiring less silicon, and thus decreasing manufacturing costs and power usage. As these attributes are valuable to chip manufacturers, most high associativity caches are implemented using the PLRU scheme.

Implementing a PLRU cache is accomplished by organising elements in a binary tree. Cache lines are stored on the leaves of the tree. Each node of the tree contains an additional bit of information that acts as a pointer. The pointers are used to approximate the least recently used element. This is accomplished by setting each pointer to point away from a memory location being updated during a touch operation. Further, to determine the element to evict, the pointers are simply followed and the element being pointed at is evicted. Hence the touch and evict operations can be defined specifically as:

- $EvictAndReplace(memoryLocation)$: Take the path indicated by the pointers from the root of the tree to the indicated leaf; store the new memory location in the cache line on the indicated leaf.
- $Touch(memoryLocation)$: For each pointer on the path between the root of the tree and the memory location indicated, set the pointer to point away from that path.

These behaviours are demonstrated in Figure 1, which shows a request for a memory location not in the cache, and hence results in an eviction operation (evicting the element which the pointers point to) followed by a touch operation (pointing the pointers away from the new cache element).

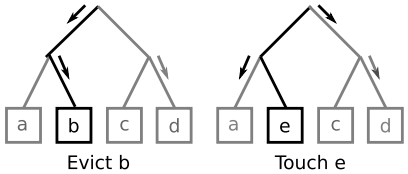

Figure 1: Showing the behaviour of a PLRU cache by accessing the element e, which evicts b

One slight complication in this description is the handling of invalid cache lines (the state of a cache line which contains

no information, such as when the cache is initially turned on). Two policies exist in this case *sequential fill* and *tree fill* [17]. In the tree fill scheme, the pointers on the nodes of the tree always determine which cache line should be evicted. However, in the sequential fill scheme, if the cache contains one or more invalid cache lines then the first invalid cache line (using an arbitrary ordering) is selected. Sequential fill can cause complications in analysis as if it were not for this behaviour, left and right subtrees of a node in the PLRU tree containing the same abstract representation could be considered equal in all cases. With sequential fill, left and right subtrees cannot necessarily be considered equal if they contain two or more invalid cache lines. Due to the complications of sequential fill, this paper only considers the case of tree fill caches.

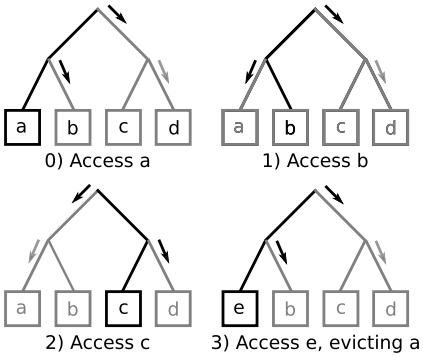

Figure 2: Illustrating the fastest a memory location can be evicted from a PLRU cache

The behaviour of a PLRU cache means that from each access to a cache element, it is guaranteed that the element will persist for at least $log_2(cache\_size) + 1$ accesses [19]. The worst case scenario that is used to find this bound is illustrated in Figure 2. The worst case can be proven by noting the fact that a memory access can only set a single pointer on the path to the required value; specifically, the pointer where the path to element a and the path to the element being accessed diverge. For all other pointers on the path to element a, the pointers will by definition be set to point away, as the paths are the same. Hence, to get all pointers on the path to element a to point to a, at least $cache\_height = log_2(cache\_size)$ memory accesses are necessary. Finally, an additional memory access is required to evict a, and hence the bound is $log_2(cache\_size) + 1$.

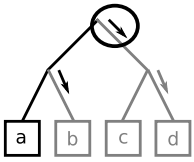

Figure 3: As an access to a sets the circled pointer, b can be unintentionally protected

Unfortunately, unlike in the case of an LRU cache where a fixed number of misses guarantees an element is evicted, there is no similar bound for the PLRU cache. This was illustrated by Berg et al. [3], who showed that it is trivial to construct a case where an element in the cache can still be resident in the cache after an arbitrarily high number of misses. This is illustrated in Figure 3, and exploits the tree structure of the PLRU cache. Specifically, where cache lines a and b reside in the same subtree, frequent accesses to cache line a will protect cache line b from eviction, by setting the pointers in the common part of the path of a and b to point away from b.

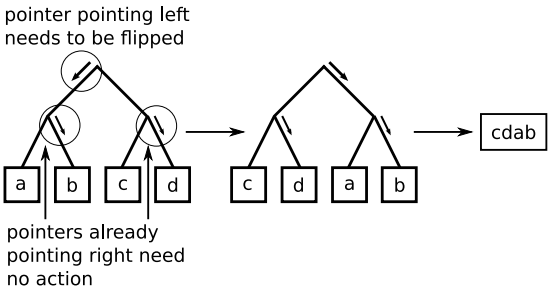

Figure 4: Assigning a name to a tree fill PLRU cache

One of the interesting properties of a PLRU cache is that multiple cache states can exhibit the same logical behaviour. This can be observed by swapping subtrees at the same level and flipping the corresponding pointer; this does not change the cache lines being pointed at, but does change physical locations[2]. To determine which states exhibit the same behaviour, a current technique, referred to as *cache naming*, swaps the left and right subtrees of any nodes of the tree which are pointing to the left, as well as flipping the direction of the pointer, as illustrated in Figure 4. By placing a cache state in this form, all cache states that exhibit the same logical behaviour will be equal, and hence duplicates can be easily detected and discarded. This scheme also enables a compact name to be assigned to the cache state, which is simply the values of the cache lines read from left-to-right, as all pointers are known to point in a single direction. As cache naming reveals the logical behaviours of the PLRU cache, it can be used as a basis for implementing collecting semantics for PLRU [12].

Initial work on providing a Must analysis of a PLRU cache was carried out in 2003 by Heckmann et al [19], and further elaborated on by Grund and Reineke [21] in 2008. As previously stated, any cache element requires at least $log_2(cache\_size) + 1$ memory accesses from its last memory access to be evicted. Hence, it can easily be inferred that a Must analysis on an LRU cache of size $log_2(k) + 1$ elements will provide a sound estimate of the elements that would be in the PLRU cache. The main problem with this approach is that it is a partial analysis which does not scale well with the size of the cache. Hence, while a cache of size 4 can have 3 elements analysed by this approach, doubling the cache size to 8 results in only 4 elements of the cache being analysed. Further, this approach does not yield any useful data about cache Misses, and hence cannot provide a May analysis.

More recently, in 2010, Grund and Reineke [17] provided an improved PLRU cache Must analysis, by utilising a metric named subtree distance. Subtree distances quantify the link between elements in the cache. By approximating the subtree distance of elements within the cache to be either maximal or non-maximal, the analysis is able to have some knowledge of how access to specific elements affect other elements, thus enabling the exclusion of additional elements from eviction when compared to the previous work.

[2]This does not hold in the case of a sequential fill cache containing invalid cache lines, as in sequential fill the physical position of an invalid cache line is significant.

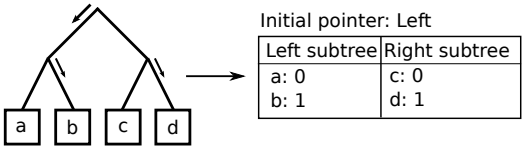

Figure 5: Mapping a PLRU cache state to Grund and Reineke's representation

The implementation of Grund and Reineke's approach considers abstract cache states with 3 components: the root pointer and representations of the left and right subtrees. Elements in the subtrees are grouped by the maximum number of pointers that could be pointing to each element in the subtree, as in Figure 5. When the maximal number of pointers reaches $log_2(cache\_size)$ these elements have been considered for eviction, and hence fail the Must test.

By analysing the root pointer of the cache tree separately, Grund and Reineke's analysis effectively decreases the height of any cache trees to approximate by 1. This results in the good results seen by Grund and Reineke for a 4-way cache [17], as the cache trees approximated have height 1, and hence behave identically to an LRU cache. However, while using the maximal number of pointers pointing to a cache element is a better approximation than using the Must analysis of an appropriately sized LRU cache, the same problem of scalability applies. Grund and Reineke's results for an 8-way cache indicate that only 6 elements of the cache can be analysed, as opposed to the entire cache in the 4-way case. This leads Grund and Reineke to conclude that their technique is limited to analysing $2log_2(cache\_size)$ elements, and hence this is still only a partial analysis. Further, the approach still does not yield any useful information for a May analysis.

3. LOSSY COMPRESSION

As mentioned before, Abstract Interpretation implements a form of Lossy Compression. In the previous example, Grund and Reineke's analysis of the PLRU cache [17], a form of lossy compression has effectively been used, in that information deemed to be important is kept while discarding information deemed less important. However, in order to devise an analysis based on the principles of lossy compression [26], it is first necessary to examine how it is relevant to the problem faced. As previously noted, while every piece of information is important, quite often some pieces of information are more important than others. Lossy compression takes advantage of this principle to discard information that is of low value to a particular result.

When applying lossy compression, the value of discarding information is twofold. Firstly, information that is discarded does not have to be stored, and this increases compression. Secondly, it is possible that the information to be discarded can actually be substituted instead [26]. If the information is substituted, then patterns can be introduced into the information stream that can be exploited by a lossless compression technique. This can result in even less information being stored than by simple discarding; by introducing patterns that a compression algorithm can effectively exploit, large sections of the stream only have to be stored a single time, thus further reducing the amount of information to store. However, determining the value of information to discard via lossy compression is highly dependent on the specific application. The next section gives an overview of determining the value of information in MP3 compression.

3.1 Overview of Lossy Compression in MP3

Perhaps the most well known example of a specific application of lossy compression is Audio compression, such as MPEG Layer 3 (MP3) [5]. Audio compression utilises an effect called psychoacoustic masking for compression, in addition to the quantisation and the Modified Discrete Cosine Transformation (MDCT).

Psychoacoustic masking describes the limits on human hearing, and specifically when the presence of one frequency of sound masks another. First conceived of in 1979, psychoacoustic masking was independently developed by Schroeder [22] and Hill and Krasner [20], it was not until later that psychoacoustic masking became a viable technique, with implementations of psychoacoustic codecs being well publicised in 1988 [4]. Two forms of psychoacoustic masking are utilised in MP3: simultaneous masking, which describes when multiple frequencies cannot be perceived at the same time, and temporal masking, which describes when sound cannot be heard around a temporally local louder noise. When information in the audio signal is identified as being imperceptible according to one of these effects, it is of no value and hence can be discarded without consequence. Similarly, frequencies above and below the range of human hearing can also be discarded. In the case of information that is discarded, an MP3 decoder replaces the sound with white noise of an appropriate volume, as this is less perceptible to the human ear than silence.

During the quantisation [15] phase of compression, the analogue signal is converted to digital. Due to the nature of sampling [24], lower frequency sounds can be described using fewer samples than higher frequencies, which leads to compression by varying the number of samples with respect to the input frequency. The final technique used is the MDCT transform, which represents a given signal as an infinite series of cosine functions [1]. Once transformed by MDCT, cosine functions which have a low weight can be discarded, as these contribute least and therefore have lower value. Therefore, MDCT transformed data can be compressed without significant reduction in quality.

As the compression can be parameterised by many variables, the size of the compressed state space is much larger than the uncompressed state space, as multiple compressed outputs can represent the same uncompressed data, albeit with varying levels of quality. However, as only a single compressed output is required, this does not detract from the applicability of the technique.

3.2 Application of Lossy Compression to WCET Modelling

Having detailed an example of lossy compression in the MP3 format, we now provide a general outline of how a more formal approach can be used to find appropriate simplifications. This can be done according to the following steps, with examples taken from the use of lossy compression in MP3:

1. *Types of Information*: Firstly, it is necessary to identify the distinct types of information within the system being modelled. Types of information are determined by constructing the minimal alphabets needed to represent unambiguously all data within the concrete system.

- Example: finding frequencies in a sound.

2. *Value of Information*: After types of information are found, it is necessary to argue for the value of each type of information as used within the system. This can be accomplished by considering the use of the information within the system. Such an argument should take into account the frequency of use of the information, with less frequently used information being less valuable, the amount of information stored in each instance of data, and the consequences of the information being inaccurate. Simple experimentation, by trivially discarding information of a certain type, may be useful here to provide evidence that an argument is correct.
   - Example: Perceptibility of frequencies in a sound at a given time due to hearing range or psychoacoustic effects.
3. *Overall Strategy*: Having decided on which types of information are least valuable, the next step is to decide how much information should be discarded by the lossy compression. All data of no value should automatically be discarded, and in addition enough low value data also needs to be discarded to make the analysis tractable. These decision on what to discard will then shape the choices in the remaining steps.
   - Example: Strategy of discarding all imperceptible frequencies when they occur, approximating the remainder using MDCT.
4. *Representation*: Once an overall strategy has been determined, a suitable representation should be found. The representation used should be picked such that it is computationally efficient to discard any information marked for removal by the strategy.
   - Example: Represent sound by applying quantisation and MDCT to the source.
5. *Approximation Operator*: Next, an approximation operator must be defined, which successfully implements the discarding of low value information.
   - Example: Discarding imperceptible frequencies and removing low value components of the MDCT transformation.
6. *Recovery Strategy*: Finally, if information that has value to the analysis is discarded, it is necessary to implement a recovery strategy to cope with the information loss. Such a strategy has to ensure that the property of soundness still holds in the analysis, despite information having been discarded.
   - Example: Replacing sounds which have been discarded with white noise of an appropriate volume during playback.

Following these steps may increase the size of the state space by introducing a notion of uncertainty into the representation. Provided that this is counteracted by visiting fewer states during the analysis, increasing the size of the state space does not in itself present a problem.

Having outlined an overall strategy for devising new analyses with a more formal approach based on lossy compression, the next section demonstrates how these methods can be applied to analysing the PLRU cache.

4. LOSSY COMPRESSION FOR THE PLRU CACHE

The first step is to determine the different *types of information* used to represent a PLRU cache. A type of information is defined by the minimal alphabet that is needed to represent it. Information using a different alphabet is counted as a different type. In the system of a tree-fill PLRU cache, there are two quantities that affect this encoding: the number of cache ways N, and the set of possible memory blocks that the cache could contain, L. With this information, it is possible to identify three distinct types of information:

- *Pointers*: The pointers on the nodes of the cache tree. These are represented by the alphabet of $\{0, 1\}$, and hence consume a single bit of information. In total, for an N-way cache, there are $N-1$ pointers, meaning $N-1$ bits of information are used for their representation. Pointers are used to determine which cache line to evict.
- *Cache Lines*: Each cache line contains either a memory block, or is invalid. Hence they are represented by the union of the set of L possible memory blocks and a single invalid state. As each cache line is drawn from the pool of possible memory blocks, for a cache of size N the number of possibilities is given by the binomial coefficient ${}^{N}C_{|L|} = \frac{N!}{(|L|-N)!}$. Cache Lines are used to determine if a given access is a hit or a miss.
- *Tree Structure*: The tree structure describes the position of each cache line and pointer within the cache, and hence is represented by an alphabet of orderings. The tree structure comprises two parts: the ordering (or permutation) of pointers (which, given $N-1$ pointers, has a size of ${}^{N-1}P_{N-1} = (N-1)!$) and the ordering (or permutation) of cache lines (which, given N cache lines, has a size of ${}^{N}P_{N} = N!$), and therefore has an alphabet size of $(N-1)!N!$. However, in most representations the tree structure will be implicit, as both pointers and cache lines will be represented with an implicit notion of ordering (e.g. as in a list). The tree structure is used in both evictions and touch operations, to determine which cache line to evict/touch respectively.

Of these items, the pointers and tree structure control the discrepancy between physical and named states of the cache. As previously mentioned, flipping subtrees such that all pointers point in the same direction can be used to assign a named cache state to a physical cache state. The information lost in this transformation is the distinction between different physical states which have the same logical behaviour, which clearly has zero value with respect to the analysis.

Considering the types of information, it is possible to evaluate their relevance with respect to the PLRU cache analysis algorithm. It is possible to divide cache analysis into three steps: a classification step, which determines if an access is a hit, miss or uncertain, an evict step that determines possible successor states in the event of a miss, and a touch step that sets the pointers in the cache appropriately, and updates the tree structure. Each of these steps uses information, and hence can be affected by uncertain information. Classification uses the contents of the cache; in the case that this

is uncertain, classification will be unable to determine cache hits or misses. Similarly, Eviction is degraded if a pointer is unknown, as both possibilities an unknown pointer represents must be considered, and hence multiple successor states will be produced. However, it is also important to note that these steps also modify the cache state, and hence can remove uncertain information as well. An eviction operation can potentially replace an unknown memory block, while a touch operation can overwrite unknown pointers.

The cache lines of a cache state are used to determine if a cache access would be a hit or a miss; this in turn determines whether or not an eviction should occur. Pointers are used to determine what element of the cache should be evicted, if an eviction is necessary. It can therefore be inferred that the information presented in the cache lines is more valuable than the information in the pointers, because for each memory access it is necessary to classify the memory access, whereas evictions may or may not happen based on the result of the classification. Further, in actual use cache hit rates are engineered to be high, as the penalty for a high number of misses is severe; this is illustrated by Cantin and Hill [8], who show that for the SPEC CPU 2000 benchmark the miss rates for LRU associative caches typically range between 10% and less than 2%, depending on the size of the cache. Assuming that this represents acceptable performance for a cache, it follows that for every evict operation that occurs, between 10 and 50 classification operations occur. This leads to the conclusion that information used during classification (the contents of the cache lines) is much more valuable than information used during eviction (the contents of the pointers). In the case that this does not hold, and evictions happen frequently, then discarding pointers will cause pessimism; however, this pessimism will only occur when the cache is largely ineffective (due to a high miss frequency). Therefore, in the case where discarding pointers introduces significant pessimism, there is an argument that the use of the cache is ineffective anyway.

It is also important to consider that any operation will result in some information being overwritten; if uncertain information introduced by lossy compression is overwritten, then the lossy compression will not have an effect on the precision of the analysis. Further, data which is overwritten provides a natural form of recovery from the lossy compression. Hence the likely frequency of overwriting data is a consideration in determining which data should be discarded by compression. As a touch operation is performed on each access, touch operations occur with the same frequency as classification operations, which is expected to be much greater than the eviction operation. The logical conclusion of this is that as the pointers of the cache are more frequently overwritten than the cache lines or tree structure, the impact of a pointer containing unknown information will be less than that of unknown cache lines or tree structure.

Finally, it is necessary to consider the number of successor states that need to be considered to handle the impact of uncertain information. In the case of cache lines, an uncertain cache line results in at most 2 successor states - the memory block is either in the cache or not. For pointers, if all pointers are uncertain then one of N memory blocks could be evicted, leading to N successor states. Similarly for the tree structure, an access to a memory block whose location in the cache is uncertain will result in N successor states, as the memory block could reside in any cache line.

	Usage freq.	Overwrite freq.	Maximum Uncertainty impact
Pointers	Low	High	N on evict
Cache Lines	High	Low	2 on classify
	(can trigger pointer and tree structure uncertainty)		
Tree Structure	V.High	High	N on touch

Table 1: Properties of Information in a PLRU cache

Summarising this information (Table 1) it can be concluded that pointers are a useful target for lossy compression. The tree structure is not an appropriate target because it is used in every operation and uncertainty in the tree structure results in a large number of successor states. Cache lines are argued to be unsuitable as they are used frequently, overwritten infrequently, and an uncertain cache line can potentially trigger uncertain pointers and tree structure to generate additional successor states. While the potentially large number of successor states might appear to make pointers an undesirable target for compression, their expected low usage combined with a high overwrite frequency mean that in typical usage this is not expected to cause a problem.

5. FULL TREE ANALYSIS

A simple implementation of collecting semantics for PLRU finds logical states by using pointers to rearrange the tree structure so that all pointers point in the same direction, and is referred to as *cache naming*. Cache naming is unsuitable as it becomes complex to check that two cache states have the same tree structure, as pointers impact the cache name substantially, as previously seen in Figure 4. Instead, an alternative method of transforming cache states is necessary, such that for cache states with the same cache lines it is easy to determine if they also have the same tree structure. This can be accomplished by using the cache lines to manipulate the tree structure, as opposed to the pointers, and will be referred to as the *cache signature*. As our analysis preserves the tree structure of the cache, it is referred to as *Full Tree Analysis*.

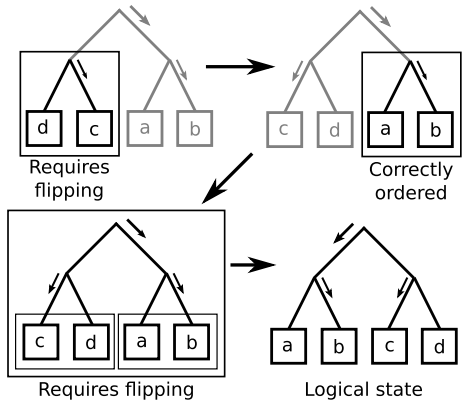

Figure 6: Illustrating how to find a cache signature, using alphabetical comparison, by moving from a physical state to an appropriate logical state without using pointers

The method of computing the cache signature, using cache lines to manipulate the tree structure, involves recursively sorting the cache tree such that for each node of the tree, the leftmost element of the left subtree is less than the leftmost element of the right subtree, using a given ordering (assuming that no memory block can be in the cache twice). With

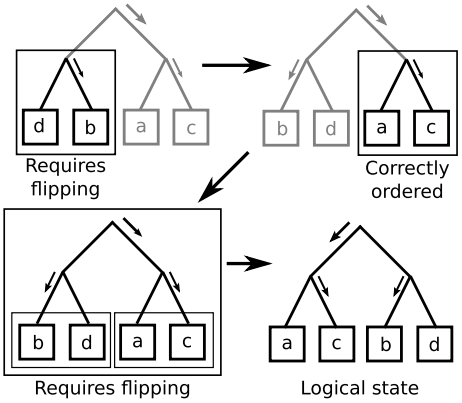

Figure 7: A second example of finding a cache signature. As the tree structure differs from the first example, the logical state reached is significantly different.

caches in this arrangement, as with the arrangement given by cache naming which uses pointers, it is trivial to check if two caches containing the same cache lines have the same tree structure. This can be done by simply reading the elements of the cache in left-to-right order. As pointers are not used to compute this ordering, the only way for two cache states containing the same memory blocks to have a different ordering of their elements is for their tree structures to differ. Hence it accomplishes the goal of determining when two cache states have the same cache lines and tree structure, and thus enables pointer information to be analysed.

An example of this is given in Figures 6 and 7, using an alphabetic ordering on the memory blocks a, b, c, d contained in the cache states. In Figure 6, first the deepest subtrees are sorted, resulting in the left hand subtree being flipped but the right hand subtree remaining the same. Then the next deepest subtrees (in this case, the entire tree) is sorted in the same manner, by comparing c and a; as $a < c$ the subtrees are swapped, resulting in the signature state a, b, c, d. Figure 7 demonstrates the same sequence of actions, but on a cache with a different tree structure. The resulting logical cache state is significantly different, with the final ordering of elements and hence the signature changed; crucially however, when using this method the only way to obtain such a difference is through cache states with different contents or tree structure - the contents of the pointers only affects the pointers in the signature state, and not the positioning of memory blocks in the signature state.

An additional advantage of removing the impact of pointers on the positioning of elements in the signature states is that the signature states become more efficient to update than named states. This is because if no eviction/replacement is carried out, a signature state remains in its signature representation. By contrast with the cache naming approach, as the touch operation changes pointers, then subtree flipping must occur. Hence, by removing the impact of pointers, the speed of analysis is increased.

Our method for lossy merging allows each pointer to occupy three states: left-pointing, right-pointing, and uncertain. When performing merging, each cache state is first transformed into its logical (cache signature) representation by applying the sorting method outlined above. Then, when multiple cache states have the same cache content and tree structure, determined by reading the contents from left to right, these cache states are merged. Each pointer in the merged cache state is set to be left (right) pointing if for all signature states being merged, the pointers in the same location are all left (all right) pointing. Otherwise, the pointer is set to be uncertain, as it must represent both values.

In order to cope with the unknown pointers, the analysis must know how to handle these during eviction. As an unknown pointer represents one of two possible values, the correct way to handle this is to consider both possible outcomes. This results in the eviction operation returning a set of successor states. As the input to the touch and classification operations is unaffected by the compression, these functions behave as they would do in a normal collecting semantics approach.

The additional compression obtained by using this merging strategy can be found by observing that as pointers no longer have an impact on the analysis, then at most 2^{N-1} states in the collecting semantics can be represented by a single state in Full Tree analysis. In practice, the benefit of this can be found in multipath code, which is more likely to have different valued pointers than single path code, provided that the hit rate of the cache is sufficiently high. If the hit rate is not sufficiently high, then the likelihood of pointers which have been discarded being used is greater, and hence additional uncertainty can be expected. However, as previously stated, programs are designed to have high cache hit rates, and therefore this property is unlikely to occur in practice.

Having outlined the changes needed to collecting semantics to implement Full Tree analysis, it follows to evaluate the effectiveness of these changes.

6. EVALUATION

Evaluation was carried out on three sets of benchmarks, using three different methods. Full Tree analysis (ft) was compared against the collecting semantics implemented by cache naming (cs), and the current state of the art, Grund and Reineke's Potential Leading Zeros analysis (plz). The first set of benchmarks are synthetic benchmarks, as compiled by Grund and Reineke [17], and are presented in Section 6.1. The second and third sets of benchmarks are multipath benchmarks extracted from the Maladärlen [10] and Papabench [14] benchmark suites respectively; for these, due to excessive memory usage, it proved impossible to provide a comparison against the plz method.

6.1 Synthetic Benchmarks

Two sets of synthetic benchmarks were applied to an 8-way fully associative cache; these benchmarks are defined in terms of memory blocks and hence the size of cache lines is arbitrary. The two sets of synthetic benchmarks evaluated are defined as follows:

- $loopK$: Subjects the analysis to a loop of the memory blocks 1 to K, with the loop repeating 16 times. This test provides a stress test to show the boundaries of the analysis.

- $randomK$: Subjects the analysis to accesses to 100 memory blocks randomly chosen from the range $[1, K]$. This test provides a sample of performance in typical non-looping conditions.

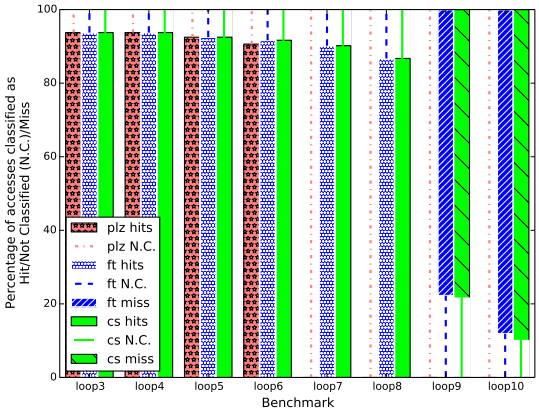

Figure 8: Results for the synthetic *loopK* benchmark

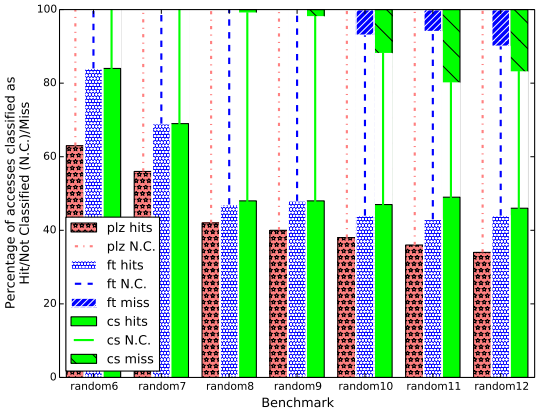

Figure 10: Results for the synthetic *randomK* benchmark

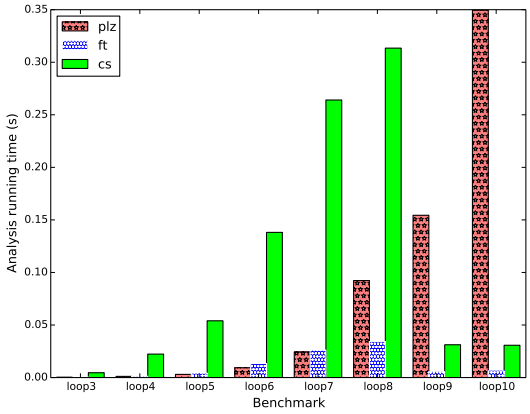

Figure 9: Analysis time for the synthetic *loopK* benchmark

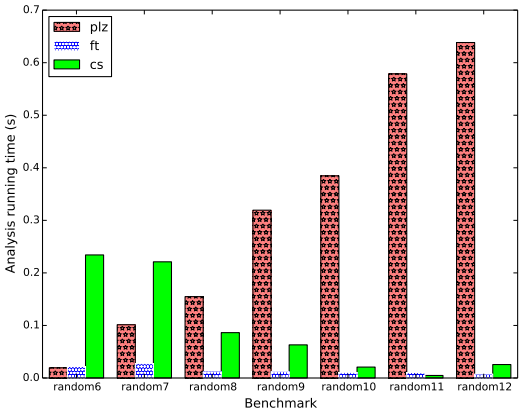

Figure 11: Analysis time for the synthetic *randomK* benchmark

Figure 8 shows the classification results for the *loopK* benchmark. This illustrates that unlike plz, ft is capable of analysing all elements in the cache. This is shown by the fact that it can predict hits for loops of size 7 and 8. Figure 9 shows the time that the analysis takes. Here, the Full Tree Analysis ft is up to 10 times faster than using the collecting semantics cs. It is also much faster than plz for high values of K, due to the additional pessimism of plz resulting in plz analysing a large number of states.

Similar benefits are seen in the *randomK* benchmarks. Figure 10 shows the classification results for the *randomK* benchmarks. Again, Full Tree Analysis ft outperforms Potential Leading Zeros analysis plz in all cases. Similarly, the time taken to reach this result is lower than that seen in cs, with the largest observed difference being approximately 10 times. Again, for large values of K, ft is much faster than plz.

6.2 Maladärlen and Papabench Benchmarks

To assess the effectiveness of Full Tree Analysis on more realistic benchmarks, the Maladärlen [10] and Papabench [14] benchmarks were used. The benchmarks were compiled for the MIPS architecture and the resulting binaries interrogated using the Heptane analyser [9] to find a control flow graph bounding all paths through the program. These graphs were then analysed using the ft and cs methods, assuming a fully associative 8-way PLRU cache with a line size of 32 bytes (for a total cache size of 256 bytes). Note a relatively small cache size was chosen so that the effects of low hit rates could be examined. The size of the control flow graphs analysed varied from less than 1KB (*cnt*) to more than 100MB (*compress*)[3], demonstrating that the technique is applicable to complex programs.

Figure 12 shows the results for selected Maladärlen benchmarks, indicated on the x-axis. For the majority, the results from ft analysis are competitive with cs. The worst results are observed in *cnt*, which exhibits some additional pessimism due to the nature of the program branches which are merged resulting in infeasible states being considered combined with the comparatively high proportion of misses. These problems also negatively impact the analysis time of ft, as shown in Figure 13 which shows *cnt* taking longer to analyse using Full Tree analysis ft than collecting semantics cs. However, it should be noted that for all benchmarks of a

[3]While *compress* is a small benchmark, the Heptane analyser is unable to provide control flow information, and hence this results in a large number of paths.

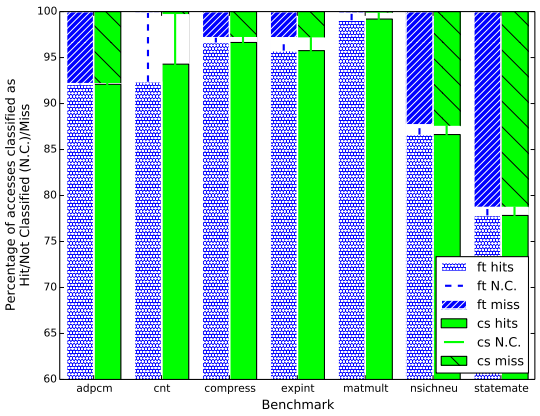

Figure 12: Results for selected Maladärlen benchmarks

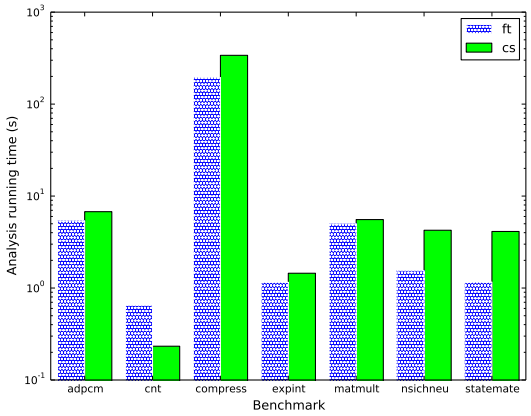

Figure 13: Analysis time for selected Maladärlen benchmarks

substantial length, ft outperforms cs by a factor of between 2 and 5 (note the log scale on the graph). In the case of cnt, even though the benchmark takes longer to analyse under ft than cs, both still perform the analysis in less than a second.

The results for the Papabench benchmarks show similar properties to those for the Maladärlen benchmarks. Figure 14 shows that when analysing the benchmark as a whole, as is the case with the $autopilot$ and $fly-by-wire$, the results are competitive with the collecting semantics. However, when the individual tasks are analysed the effect of the introduced pessimism is proportionally higher, leading to proportionally worse results. Similarly, the time taken for analysing individual tasks is worse for ft than cs, but still less than a second. For the larger benchmarks, ft is faster than cs by a factor of 5.

As previously stated, a small cache size was picked to examine the effects of low hit rates. A larger cache could be analysed by increasing the size of a cache line or moving to a set associative cache. Further, in addition to the tests presented here, a 16-way cache was also attempted. However, this was not feasible on the available hardware due to memory usage, which has been calculated to be approximately 3 orders of magnitude higher than is necessary for the 8-way

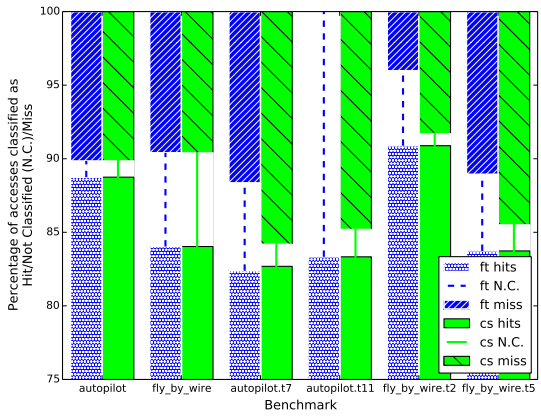

Figure 14: Results for the Papabench benchmarks

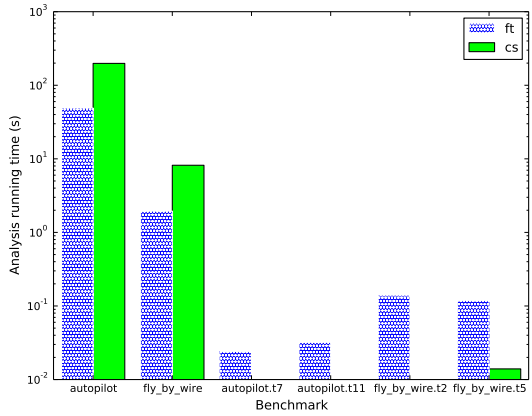

Figure 15: Analysis time for the Papabench benchmarks

cache; for the current implementation, this is thought to be approximately 8GB. As the memory usage is so high, limitations in commercial computers mean that the analysis will spend the majority of CPU resources paging memory, even if it can complete the analysis. Hence it is concluded that the analysis of a 16-way cache is only feasible with specialist hardware capable of handling large data sets.

7. CONCLUSION

This paper introduced the idea of applying Lossy Compression to the collecting semantics of a given component, with the aim of providing a general and powerful approach to formulating WCET analyses. Specifically, existing techniques from abstract interpretation were compared against techniques used in lossy compression to find similarities. These similarities were used as the basis of a generic approach to formalise creating a simplification that can be used in abstract interpretation. Even though this approach may increase the size of the state space for the analysis, when correctly applied the number of states visited during analysis should decrease, as is the case with correctly applied abstract interpretation.

While any analysis derived by using Lossy Compression could potentially be found by other methods, utilising such

a formal approach grants access to a structure and process which can aid in understanding the effects of approximating the various quantities in the system under analysis. Thus lossy compression provides an effective means of reducing the difficulty involved in finding appropriate approximations.

To validate this approach, a PLRU cache analysis was designed. Full Tree analysis was shown to outperform the previous state of the art technique with respect to both the accuracy of analysis and the runtime required, in addition to also providing a May analysis. In comparison to the collecting semantics, Full Tree analysis was shown to be faster on substantially sized realistic benchmarks, while providing comparable accuracy.

In [16] we have also applied the Lossy Compression approach to provide analysis for caches with an evict-on-miss random replacement policy [13], [2].

Acknowledgement

This work was partially funded by the EU FP7 Integrated Project PROXIMA (611085).

8. REFERENCES

[1] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, 100(1):90–93, 1974.

[2] S. Altmeyer and R. I. Davis. On the correctness, optimality and precision of static probabilistic timing analysis. In *17th Design, Automation and Test in Europe Conference (DATE)*. EDAA, 2014.

[3] C. Berg. Plru cache domino effects. In F. Mueller, editor, *6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum f"ur Informatik (IBFI), Schloss Dagstuhl, Germany.

[4] K. S. D. Brandenburg. Ocf: Coding high quality audio with data rates of 64 kbit/sec. In *Audio Engineering Society Convention 85*, 11 1988.

[5] K. S. G. Brandenburg. ISO/MPEG-1 audio: A generic standard for coding of high-quality digital audio. *J. Audio Engineering Soc*, 42(10):780–792, 1994.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Infinite Computing*, 98(2):142–170, 1992.

[7] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison Wesley, fourth edition, May 2009.

[8] J. F. Cantin and M. D. Hill. Cache performance of SPEC 2000 CPU. `http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data`, May 2003. Accessed on 15th August 2013.

[9] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *13th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 37–44, 2001.

[10] Contributors. Maladarlen WCET benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html. Accessed on 1st September 2013.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[12] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.

[13] R. I. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 168–179. IEEE, 2013.

[14] Nemer F., Cassé H., Sainrat P., Bahsoun J., and Michiel M. Papabench: a free real-time benchmark. In *In WCET '06*, 2006.

[15] A. Gersho and R. M. Gray. *Vector quantization and signal compression*, volume 159. Springer, 1992.

[16] D. Griffin, B. Lesage, A. Burns, and R. I. Davis. Static probabilistic timing analysis of random replacement caches using lossy compression. In *RTNS '14: Proceedings of the 22nd International Conference on Real-Time and Network Systems*, Versailles, France, 2014. ACM, New York, NY, USA.

[17] D. Grund and J. Reineke. Toward precise PLRU cache analysis. In B. Lisper, editor, *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 28–39. Austrian Computer Society, July 2010.

[18] J. Handy. *The cache memory book*. Morgan Kaufmann, Burlington, Massachusetts, USA, 2nd edition, Jan 1998.

[19] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[20] M. A. Krasner. Digital encoding of speech and audio signals based on the perceptual requirements of the auditory system. Technical report, DTIC Document, 1979.

[21] J. Reineke and D. Grund. Relative competitiveness of cache replacement policies. In *SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 431–432, New York, NY, USA, June 2008. ACM.

[22] M. R. Schroeder, B. S. Atal, and J. L. Hall. Optimizing digital speech coders by exploiting masking properties of the human ear. *The Journal of the Acoustical Society of America*, 66(6):1647–1652, 1979.

[23] C. E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27, 1948.

[24] C. E. Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[25] M. Usher. *Information Theory for Information Technologists*. Macmillan Publishers Ltd, 1984.

[26] J. Watkinson. *Compression in Video and Audio*. Focal Press, 1995.