7 Memory Awareness

Due to the involvement of massive data and the growing size of trained models, most machine learning techniques are memory intensive. As one of essential components in the von Neumann architectures widely used nowadays, memory is a well-known bottleneck on the execution time, particularly due to the "Memory Wall" problem. That is to say, the access time of memory is way larger than the processor cycle time. In addition, the energy and power consumption required by the memory are known to be significant in the overall system. On embedded systems, which is the focus of this book, such design constraints are amplified and impose great challenges for machine learning techniques. Although the emerging non-volatile memories appear to be promising because of their attractive features, e.g., low leakage power, high density, and low unit costs, they also bring up new design constraints like higher error rates, which might degrade the performance of machine learning techniques. To this end, several optimization and architecture-aware approaches have been proposed to improve the usage of memory and enhance the reliability of learning algorithms.

In this chapter, several techniques are briefly introduced to tackle some of the aforementioned issues related to memory. By leveraging the application-specific knowledge, we demonstrate that the memory footprint can be effectively reduced (see Section 7.1). Given learning models, we can further optimize the memory layout proactively in the model implementation to favor the underlying cache memories with a probabilistic perspective (see Section 7.3). Last but not the least, learning models can be reliable with unreliable memories if we take bit errors into account during the training phase (see Section 7.2). Overall, this chapter tends to suggest that the design constraints of underlying memory can be handled in a post-optimization fashion, within the implementation of learning models, or even earlier at the training phase. The insights presented in this chapter should remain highly relevant in upcoming years, and become more important for future applications along with emerging memory technologies and their new design constraints.

https://doi.org/10.1515/9783110785944-007

7.1 Efficient Memory Footprint Reduction

Helena Kotthaus
Peter Marwedel

Abstract: This section discusses optimization approaches for the efficient memory footprint reduction of machine learning algorithms that are written in the GNU R programming language. The presented optimization strategies target the memory management layer between the R interpreter and the operating system and reduce the memory overhead for large data structures by ensuring that memory will only be allocated for memory pages that are definitely required. The proposed approaches use additional information from the runtime environment, e.g., the short-term usage pattern of a memory block, to guide optimization. The evaluation is based on statistical machine learning algorithms. When the memory consumption hits the point that the OS starts to swap out memory, optimization strategies are able to speed up computation by several orders of magnitude.

7.1.1 Motivation

In order to execute machine learning algorithms on resource-constrained devices, it is important to make efficient use of the available resources. These resources include processors (including runtime), memories, communication bandwidth, and energy. This book includes sample optimization algorithms aiming to achieve resource efficiency. In particular, Chapters 6 to 9 present such sample optimizations. The current section demonstrates the optimization potential memories as resources. Ideally, memories have an infinite capacity, but their size can have a relevant impact on the applicability of certain techniques. This is especially true for resource-constrained embedded systems. The current section focuses on the efficient use of memories for machine learning algorithms written in the R language. The R language is used for many machine learning applications and, therefore, it is considered here. As shown in [387, 503], the R environment has several drawbacks leading to slow and memory-inefficient program execution. In R programs, most data structures are vectors. When the size of a vector is above a certain threshold, the R interpreter allocates a large vector. For each large vector, a dedicated block of memory is allocated, potentially spanning multiple pages. These pages, even when unused, take up memory. When the amount of memory required for computations exceeds the physical memory available to the application, the execution is drastically slowed by frequent page swaps that require I/O, a phenomenon also known as “thrashing”. The performance penalty due to thrashing might render complex computations entirely infeasible.

The current contribution is based on the work of Kotthaus et al. [383, 385, 386]. Section 7.1.2 provides a survey of related work and explains the fundamentals of R's memory management. Section 7.1.3 discusses the page-sharing strategies for efficient memory utilization of R machine learning algorithms. Finally, Section 7.1.4 presents the evaluation results and concludes with a summary.

7.1.2 Related Work and Fundamentals: Memory Footprint Reduction and the R Environment

Related Work - Memory Footprint Reduction The memory optimizations presented in Section 7.1.3 work on a layer between the R interpreter environment and the OS. This enables the optimization of arbitrary R applications, especially memory-hungry machine learning applications, with only small modifications to the R interpreter and without requiring application changes. Thus in the following, the related system-level approaches for reducing memory utilization will be discussed.

In general, related work on utilizing main memory more efficiently can be categorized into two groups: memory compression approaches, often influenced by embedded systems resource constraints, and memory deduplication, which is mostly used in virtualization.

Memory compression tries to reduce the swapping activity of a system by compressing memory contents instead of swapping pages to the secondary storage. Compression approaches share the drawback that a significant amount of processor time is spent on compressing and decompressing memory contents.

By contrast, memory deduplication reduces the memory overhead by mapping virtual pages with identical contents to a single physical page. This is often beneficial in virtualized environments where large amounts of read-only memory, such as shared libraries, are used in multiple virtual machines [626]. However, deduplication can introduce significant computational overhead, since the contents of pages have to be scanned periodically in order to identify pages with identical content. An often used implementation of deduplication that has been the subject of multiple improvements is available in Linux as the *Kernel Samepage Merging* (KSM) [22]. KSM has also been optimized in [133] by introducing a classification scheme based on access characteristics, comparing only pages within the same class to reduce the overhead of page scanning. A memory trace-based evaluation of different deduplication and compression approaches is presented by Deng et al. [169], showing that deduplication yields better results than memory compression.

Sharing memory pages within a single process appears to be a rarely-used concept: on Linux, it is automatically used to map a set of newly allocated virtual pages to a single physical page filled with null bytes. This can cause performance issues in high-performance environments since each write to any newly allocated page will trigger a page fault. Here, an enhancement by Valat et al. [678] was proposed that avoids

unnecessary page removal when the application knows that it will overwrite a page in the near future. A language-level version of this *copy-on-write* technique, operating on objects instead of memory pages, is sometimes implemented using reference counters [665]. The R language also implements a copy-on-write scheme. Here, the complete object (potentially spanning multiple pages) is copied when it is modified, resulting in page duplications for partial modifications.

OS level optimizations lack knowledge about the specific memory behavior of the runtime environment. Although some information can be used to improve the time needed to detect duplicates, the application-specific knowledge of why the data was copied in the first place is ignored. By contrast, the memory optimization presented in Section 7.1.3 employs specific knowledge about the interpreter state to reduce the number of pages that need to be scanned for identical content and *proactively* avoids the main sources of identical-content pages from object allocation and duplication by optimizing the copy-on-write mechanism for partial object modification.

Fundamentals – The R Environment The *lifecycle of an object*, (e.g., a vector data structure) in the R runtime environment starts with its allocation. In R, vectors are assumed to consist of a contiguous block of (virtual) memory. Depending on the size of the object, the R interpreter uses a system of multiple memory pools for vector objects with a data size of up to 128 B. For larger vectors, memory is allocated directly via the malloc C library function instead of pooling the allocations. This reduces the memory fragmentation when many small objects are created and some of them are released. The R language does not require the programmer to explicitly manage memory; a garbage collection is needed to automatically free memory. The garbage collector in R is a mark-and-sweep, non-moving, generational collector. It can be manually triggered, but it also starts automatically when the interpreter is in danger of running out of heap space.

The R interpreter ensures that an allocated object is always initialized—either by explicit initialization or implicitly by writing the results of a computation to it. After the object is allocated and initialized, it can be used as input for various R functions such as the plus operator. The fact that functions can modify an object, in conjunction with R implementing *call-by-value* semantics, means that objects need to be copied when being passed to a function. However, at this point a copy-on-write optimization is triggered: copying an object is done by merely sharing the reference; the actual copy is delayed until the object is modified (if at all). The interpreter now has two references to the same object, which may be modified later. When this modification happens, the copy process is triggered and a full copy of the affected object, potentially spanning multiple pages, is created using the interpreter-internal *duplicate* function. This is illustrated in Figure 7.1.

On the left-hand side, a large R vector object consisting of a header *H* and four pages *A* to *D* is shown both in *virtual memory* on the top (marked with dotted lines) and its corresponding allocated *physical memory* on the bottom (solid lines). On the right-hand

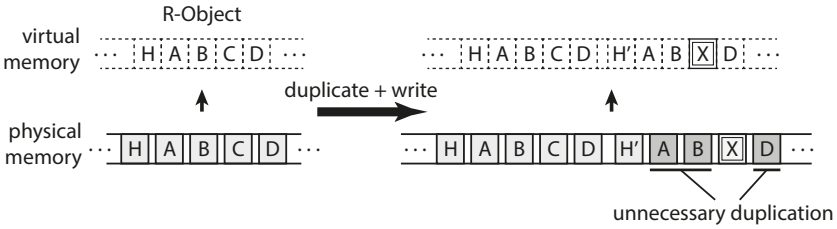


Fig. 7.1: Example of the copy-on-write mechanism in the R interpreter. R copies (duplicates) at object level instead of page level granularity [385].

side, the situation after a duplication that was triggered by a write access is shown. Now there are two R objects, shown in the virtual memory on top and their corresponding physical memory on the bottom. In one of the copies, page *C* was modified and is now marked as *X*, and the copy has its own header *H'*. Although unmodified, the R interpreter needs to use additional memory to create duplicates of pages *A*, *B*, and *D* (marked in gray) since it assumes that objects are organized as contiguous blocks of memory and thus it has to duplicate at *object-level granularity*.

The memory optimization presented in this contribution has the goal of reducing this memory overhead by copying only parts of the object, sharing the same memory pages between multiple objects as long as they are not modified. This scheme is transparent to the interpreter's memory management including the garbage collection, requiring only small changes in memory allocation and freeing, as well as in the duplicate function. This optimization will be presented in the next section.

7.1.3 Memory Footprint Reduction via Page Sharing Strategies

Different optimization strategies are combined for the efficient memory footprint reduction of machine learning algorithms implemented in the R language. The first strategy that proactively *avoids the duplication* of memory pages is based on optimizing the allocation and duplication mechanisms of the R interpreter. This approach is further refined by a second strategy using *static annotations* to reduce the optimization overhead and by *dynamic refinement* using a page content analysis for page deduplication to increase the amount of shared memory.

Page Duplication Avoidance As shown in the previous section, the R interpreter can only allocate complete objects that potentially span multiple pages. The first part of the memory optimization is based on the object allocation mechanism of R. To enable the allocation and thus the sharing of memory at *page-level granularity* instead of object granularity, a custom memory allocator is employed when a large vector has to be allocated, as shown in Figure 7.2. When the internal function of the R interpreter *allocVector*

is called to allocate a large vector, the optimized interpreter selects between the *custom allocator* to share memory on page granularity or the *default malloc* function if this is not required. In both cases, the allocated memory is accessible within the address space of the R interpreter. The custom allocator uses a memory management scheme

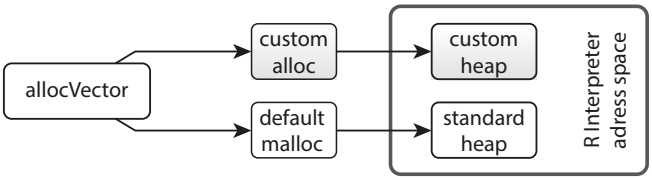


Fig. 7.2: Memory allocation scheme for dynamic page sharing [385].

similar to standard virtual memory schemes commonly used in Operating System (OS) kernels. For ease of implementation, it is completely implemented in the user space. The downside of such a user-space implementation is that it needs to replicate certain data structures that are already present in the OS (e.g., for mapping virtual to physical memory) because those OS kernel data structures are not sufficiently exposed to user space. This replication could be avoided by implementing the optimization in the Operating system kernel (cf. [383]), but this is significantly more invasive and not applicable in many environments where the user has no control over the Operating system kernel. Since the user space has no direct access to physical memory, a single file located on a RAM disk (see *custom heap* in Figure 7.2) is used.

The allocation of physical memory from this file is realized via a simple free-bitmap based allocator. The file can be dynamically enlarged if needed. Mapping physical pages into the virtual address space of the R interpreter can be accomplished by utilizing the mmap Unix system call. For changing the access permissions of these physical pages, the mprotect system call that modifies the settings of the memory management unit of the processor is employed. Besides these system calls, an additional page table is needed to enable the mapping from a virtual address to a physical address. For simplicity reasons a hierarchical page table with the same four-level structure as used by the processor is implemented. To enable the sharing of pages, the user space memory management system needs to map the same physical page to multiple locations in virtual memory. Therefore, a reference counter is required for each physical page. A reference counter greater than 1 indicates that the page is shared between multiple objects or multiple times within one object.

To avoid the zero-initialization of allocated large vector objects, a *global shared zeroed page* is utilized. This also ensures that memory is only allocated for pages that are actually written to at a later time. Figure 7.3 illustrates an example for this optimized R object allocation. Here, the custom memory allocator was asked to allocate an object with a total size of five pages. While the object has the requested size of five pages in

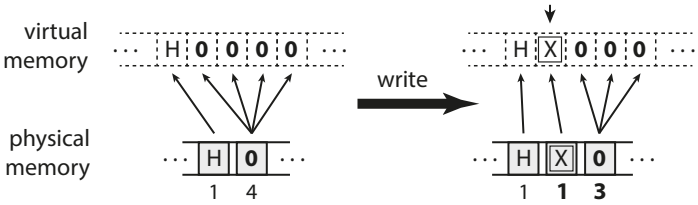


Fig. 7.3: Optimized object allocation via sharing a global zeroed page [385].

virtual memory (dotted, left upper part), physically it only consists of two pages (left lower part). Those two pages are a single non-shared page, marked with *H* for header in the beginning, followed by a shared page, marked with ***0***, called the global zeroed page. The numbers in small print below the physical pages are the *reference counters*. The zeroed page has a reference counter of 4 since it is shared four times within the allocated object (mapped four times into virtual memory). The shared zeroed page is filled with zero-bytes. The concept of prepared zeroed pages is already implemented in OS kernels. However, the standard R interpreter does not benefit from this concept since it immediately writes to all memory that it allocates for initialization. The non-shared initial page *H* is required as it will contain not just data but also the object header. The R interpreter writes this object header to the front of the allocation area. Since it will be updated frequently (e.g., during garbage collection), it is not shared between multiple objects. Since the header page *H* is mapped only once, its reference count is 1.

The R interpreter now has the illusion that it has allocated five pages of memory, even though only two pages are allocated physically. To sustain this illusion, the optimized allocation mechanism has to ensure that any write access to a virtual page that points to a shared physical page can be detected and handled. If such a write access is not handled correctly, it affects not only the intended virtual page but also all virtual addresses where the same physical page is shared. Therefore, all pages with a reference counter greater than 1 are marked as *read-only*, ensuring that a write access triggers a segmentation fault. This fault is caught by a *signal handler* that performs the unsharing of the affected page. To determine the affected physical page the handler uses the virtual address of the write access. It then allocates a new page, copies the contents of the original page to it, and replaces the page that caused the segmentation fault with the new one. The resulting situation is shown on the right side of Figure 7.3: one of the instances of the zeroed page that was written to was replaced with a new page marked with *X*. This updates the reference count of both the zeroed page and the newly allocated page. Since the new page is only mapped once, it can now be marked as read-write so that further access no longer requires special handling.

As noted, the R interpreter can only copy on the object level. Thus, if an object consists of multiple pages, parts of the copy may end up with the same content as the original (see Figure 7.1). To avoid this, the *duplicate mechanism* of the interpreter is optimized by improving the granularity of the copy from object level to page level.

While the allocation optimization avoids the immediate allocation of pages by using the global zeroed page, the duplicate optimization allows the reuse of already-allocated pages of the original object instead of allocating new pages. An example of the duplicate optimization is shown in Figure 7.4.

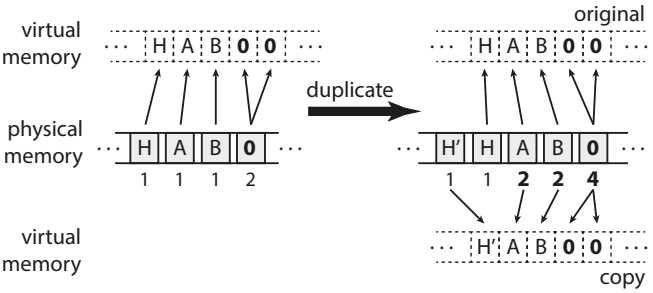


Fig. 7.4: Optimized copy mechanism on page-level instead of object-level granularity via page sharing [385].

The left side shows the situation before the duplication is shown: an object occupies five virtual pages, two of which reference the global zeroed page. Unlike the original R interpreter that would need to allocate five new pages for the copy of this object, the optimized version reduces this to a single allocated physical page. This is shown on the right side with the original object at the top and its copy at the bottom. Here, a *virtual-only copy* of the first page that contains the object header is not created, since the header of the copy is updated immediately by the R interpreter after the duplication. This would otherwise trigger an unsharing of this page. To avoid the overhead of this event, the optimized duplication immediately creates a physical copy of the header page. Most of the pages of the original object are now mapped twice in virtual memory and their reference counters are updated. Both the original and copy are marked as read-only to allow for unsharing on write access.

Overall, the finer copy granularity of the optimization enables storing both the original and copied objects from the example in just five pages of memory. By contrast, the original R interpreter would need ten pages of memory to store the same objects. Although the mechanisms of sharing pages during allocation and duplication described above always result in a valid view on memory for the interpreter, there are cases where additional overhead is caused that can be avoided by further refinements described in the next subsection.

Static Refinement via Annotations To reduce the runtime overhead caused by proactively avoiding page duplications, a static refinement consisting of two kinds of annotations is applied. The first annotation is based on the expected utilization of an

object immediately after allocation and the second annotation is based on the size of the allocated object.

The optimized memory allocation (see Figure 7.3) reduces the memory footprint by using a global zeroed page, assuming that not all pages of the allocated object will be written to immediately. However, this assumption is not always valid. For instance, (built-in) vector arithmetic functions in the R interpreter allocate a new object and immediately write to all pages of it to store their results. This would cause a segmentation fault for the first write of every page, triggering the memory allocation for all pages of the object. These segmentation faults cause runtime overhead that would not occur when allocating an object with non-shared pages.

To avoid this overhead, *annotations* are placed in the C source code of the R interpreter built-in functions where newly allocated memory is completely overwritten directly after allocation. Here, the custom allocator returns an object where every virtual page references a new physical page, so no segmentation faults will be triggered by write accesses. Although these R objects do not save memory on allocation, they still have the opportunity for later optimizations, e.g., when they are duplicated. Currently, the annotations for these *"full-overwrite"* functions need to be manually placed in the R interpreter's C source code by locating calls to *allocVector*, followed by loop structures that write to every element of the newly-allocated object. Those manually placed annotations could also be automated by a static code analysis checking for allocation calls followed by loops writing to the newly-allocated object.

The second annotation for reducing the runtime overhead incurred by the optimization relates to the size of the allocated object. The R interpreter can allocate objects with a variety of sizes, not all of which span multiple pages. The optimized custom allocator is therefore enabled only for object sizes that indicate a potential for page sharing. Here, the potential is limited for smaller objects. The first page of an object stores not just data but also the frequently modified object header that is therefore never shared. Thus R objects smaller than two pages of memory are passed to the standard, non-sharing memory allocator. This size limit could also be used as a tunable parameter to select a trade-off between memory savings and runtime overhead.

Dynamic Refinement via Page Contents In addition to the above-described static refinements, an additional dynamic refinement for increasing the number of shared pages is applied. During the execution of an R program, allocated objects are updated with the results of calculations. Those updates can result in multiple distinct pages with the same contents, which opens up the opportunity for sharing those pages. The general idea of locating identical objects in a system and saving memory footprint by reducing them to a single object is known as deduplication.

The memory optimization employs a restricted version of locating identical contents. Here, the content scan only checks for pages identical to the already existing global zeroed page. The *deduplication of zeroed pages* is illustrated in Figure 7.5. On the

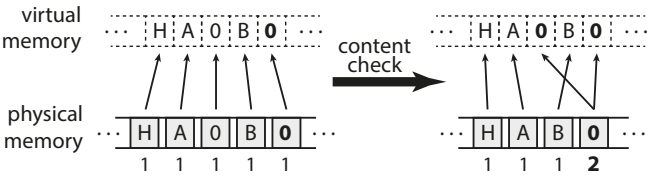


Fig. 7.5: Deduplication optimization for zeroed pages [385].

left side, the situation before the page content scan is shown where an object contains two identical zero pages. One of those pages is already mapped to the global zeroed page (shown in bold), while the other uses a separate physical page. On the right side, the situation after deduplication is shown. Here, the *content check* has detected the separate copy and mapped its virtual page to the global zeroed page, freeing the memory used for the unnecessary duplicate.

Although a general scan that is able to detect duplicated pages with arbitrary content could be applied, such a scan would incur a significant runtime overhead (e.g., due to the calculation of hash values) compared to scanning just for zeroed pages. While a scan for zeroed pages can use an early abort condition as soon as a non-zero element is found, a scan for arbitrary content would need to check the full content of all pages in the system. The overhead incurred by deduplication of zeroed pages is influenced by the frequency of the content check and by the number of pages that need to be scanned. To reduce this overhead, the scan is only activated after the completion of a garbage collection in the interpreter. This avoids scanning the pages that would soon be discarded and also provides a natural regulation mechanism for the frequency of content checks, as the frequency of garbage collection depends on the memory requirements of the executed program.

With the deduplication optimization, pages that were previously excluded from sharing the global zeroed page, in arithmetic vector operations, say, can now be dynamically shared. Thus, both the static and the dynamic refinements of the memory optimization complement each other. Details on the interaction of the refinement strategies and the general page duplication avoidance strategy can be found in a separate publication [384].

7.1.4 Evaluation: Memory Footprint Reduction Strategies

The results obtained by applying the proposed memory optimization strategies for R to real-world machine learning benchmarks are presented in this section. Both, the evaluation results related to the memory consumption and the runtime effects of the page sharing optimization strategies will be discussed.

Experimental Setup For the following experiments, a computer equipped with a 2.67 GHz Intel Core i5 M480 CPU and 6 GB of RAM, using a 64-bit version of Debian Linux 7.0 as the operating system is used. On this system, memory pages have a size of 4096 bytes. Although a larger page size than the system page size could be used for the memory optimization, the same size was chosen as it is expected to maximize the amount of memory that can be shared (using a smaller page size than the system size is inefficient since the optimization relies on the hardware Memory Management Unit (MMU) for efficient page access protection). To evaluate the proposed memory optimization approach, the memory usage and runtime of the R interpreter including the described optimizations is compared to the standard GNU R interpreter. Both the standard as well as the optimized interpreter are compiled using GCC version 4.7.2 with the default flags (-O2) selected by the build system of R version 3.1.0.

The standard memory measurement functions for user space functions in Linux measure only the virtual memory of a process. Since the optimization approach maps the same physical page multiple times into virtual memory, these functions are not sufficient for the evaluation. They are not able to measure the amount of physical memory saved since they only count every virtual instance of a shared physical page. Therefore, a separate memory measurement function was created. To measure the amount of memory allocated by the default allocator, the standard allocation functions such as malloc are overwritten with versions that track the current total amount of memory allocated and the original functions are called afterwards. For the optimized custom allocator, the number of physical pages that need to be reserved is directly tracked along with the size of the memory management data structures. With these mechanisms, the allocated physical memory can be measured accurately.

For the evaluation of the optimization, two different benchmark sets are applied. The first set is a shorter-running set of benchmarks, selected from the R benchmark 2.5 suite [274], which was originally developed to measure the performance of various configurations of the R interpreter (in the following denoted by GU) plus one additional benchmark, as listed in Table 7.1. The R benchmark 2.5 suite was also applied in other optimization approaches that focus on dynamic compilation for R [353]. To analyze if the memory optimization is also beneficial for algorithms that already try to reduce the memory footprint by using application-specific knowledge, the additional benchmark *glmnet* is included. This benchmark utilizes an existing sparse matrix optimization implemented as an R package. For accurate measurements, the iteration counts for the outer loop of each benchmark were scaled to result in a runtime of approximately 1 minute with the standard R interpreter.

The second set of benchmarks is based on a set of publicly available long-running real-world machine learning benchmarks [384], listed in Table 7.2. The choice of these classification algorithms is based on the method's popularity and the availability of its implementation. The default parameters or, if available, the implementation's internal auto-tuning process was used to configure the algorithm parameters. The input dataset is a 2-class classification problem and has a sufficiently large number of observations

Tab. 7.1: Misc Benchmark Set.

Benchmark	Description
GU/08a-1	Linear regression over a 3000 × 3000 matrix
GU/08a-2	FFT of 2 400 000 random values
GU/08a-3	Inverse of a 1600 × 1600 random matrix
GU/08a-4	Greatest common divisors of 400 000 pairs (recursive)
glmnet	Regression using glmnet on a sparse 20 000 × 1000 matrix

Tab. 7.2: Machine learning benchmark set.

Benchmark	Description
ada	Boosting of classification trees
gbm	Gradient boosting machine
kknn	*k*-nearest neighbour classification
lda	Linear discriminant analysis
logreg	Logistic regression (binary classification decision derived using a probability cutpoint of 0.5)
lssvm	Least-squares support vector machine
naiveBayes	Naive Bayes classification
randomForest	Random classification forest
rda	Regularized discriminant analysis
rpart	Recursive partitioning for classification trees

to achieve accurate results. The machine learning benchmarks were configured to use a 20-fold cross-validation. The size of the input dataset (15 000 samples with 200 numeric features) was chosen to ensure that the runtime of the fastest algorithms is approximately one minute on the standard interpreter. To allow for a better comparison of the memory requirements, the same dataset was applied to all machine learning algorithms.

Each benchmark was executed 10 times with the standard and the optimized version of the R interpreter. The results are given as the arithmetic mean of these 10 executions. To make the results reproducible, the random number seed is selected as a fixed value placed as the first statement in each of the benchmarks. Each repetition was started in a fresh interpreter process; hence initialization costs are included in the measurements (an expected overhead on the order of one second or less). The R interpreter does not use adaptive optimizations. All system services that might interfere with the measurements were disabled. Both runtime and memory usage were measured simultaneously. For these measurements, we calculated a 95 % confidence interval and the ratio of the means using the percentile bootstrap method. We use geometric means here to reduce the influence of outliers.

Memory Consumption To analyze the benefits of the page sharing optimization techniques with regard to the memory consumption we evaluate the global peak memory usage and the average memory usage of each benchmarks. The *Peak usage* represents the maximum amount of memory that was consumed during execution of a benchmark. However, the peak memory consumption does not represent information about changing memory usage over time, since the peak memory usage may occur only for an instant of time depending on the benchmark. To gain a complete view of the memory consumption the short-term peak usage is measured in intervals of 1 second, resulting in a memory-over-time profile. The *Average usage* of memory is calculated as the arithmetic mean of these 1 second measurements and used as a second indicator to allow easier comparisons of the memory behavior.

Figure 7.6 shows the peak (*Peak usage*) and average (*Average usage*) memory consumption of each benchmark running with the page-sharing optimization. The 100 % baseline represents the standard R interpreter without optimizations. Values below this baseline indicate relative memory savings realized by the page sharing strategies. Error bars have been omitted as the confidence intervals were smaller than 0.5 % for all values. The detailed values are presented in Table 7.3, including the number of pages identified as shareable by the content check. They indicate the optimization potential of the dynamic refinement (deduplication of zero pages).

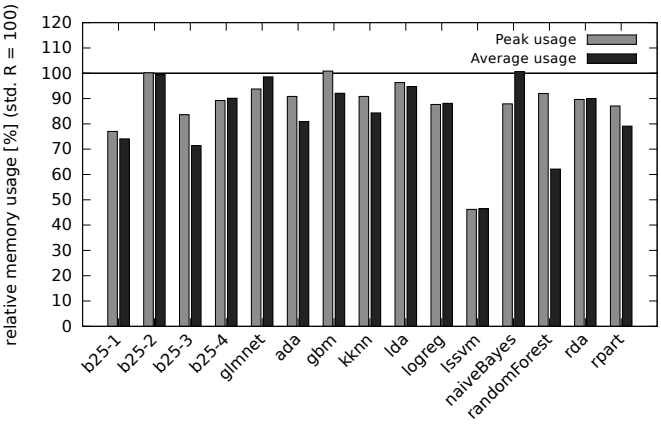


Fig. 7.6: Relative memory usage with page-sharing optimization compared with standard R (lower is better). The 100 % baseline represents the standard R interpreter without optimizations. Geometric means for the memory savings are 13.6 % for peak and 18.0 % for average memory usage [385].

The gain for reducing the peak memory usage (*GainP*) of the standard R interpreter (*StdPeak*) ranges from –0.9 % for `gbm` to 53.8 % for lssvm. However, the negative values in the columns *GainP* and *GainA* of Table 7.3 indicate that the page-sharing optimizations do not gain memory savings for three of the benchmarks. Here, the peak memory

Tab. 7.3: Memory Optimization Results: StdPeak – peak memory usage by the standard R interpreter; OptPeak – peak memory usage by optimized interpreter; GainP – relative peak memory reduction achieved by optimization; StdAvg – average memory usage by the standard interpreter; OptAvg – average memory usage by optimized interpreter; GainA – relative average memory reduction achieved by optimization; ZPG – number of zero pages found by the content check [385].

Benchmark	StdPeak [MB]	OptPeak [MB]	GainP [%]	StdAvg [MB]	OptAvg [MB]	GainA [%]	ZPG [#]
GU/08a-1	296.2	228.1	23.0	259.6	192.2	25.9	13
GU/08a-2	131.1	131.4	-0.2	128.8	128.0	0.6	13
GU/08a-3	197.2	164.8	16.4	157.7	112.6	28.6	37 919
GU/08a-4	134.2	119.7	10.8	127.2	114.6	9.9	194 892
glmnet	354.9	332.8	6.2	249.5	246.0	1.4	46 877
ada	187.2	170.1	9.1	156.0	126.2	19.1	2 031 992
gbm	191.5	193.2	-0.9	147.7	136.0	7.9	464
kknn	316.5	287.6	9.1	274.0	231.0	15.7	421
lda	216.2	208.2	3.7	184.8	175.1	5.3	20 447
logreg	213.0	186.7	12.3	184.7	162.8	11.9	955
lssvm	1365.1	631.0	53.8	820.2	381.1	53.5	3 972 699
naiveBayes	143.6	126.2	12.1	80.8	81.3	-0.6	78
randomForest	565.5	520.4	8.0	390.8	242.7	37.9	1 130 650
rda	254.1	227.7	10.4	197.0	177.3	10.0	707
rpart	144.5	125.8	12.9	130.7	103.3	20.9	56 214

consumption for two of the benchmarks (`gbm`, `GU/08a-2`) and the average memory consumption for one benchmark (`naiveBayes`) increase slightly. This is caused by the additional data structures that are needed for the internal handling of memory pages.

For `gbm`, a reduction of the average memory usage by 7.9 % (*GainA*) is achieved. For `naiveBayes` the situation is reversed: the optimization saves 12.1 % of its peak memory usage while its average memory usage (–0.6 %) is slightly increased. Since the number of pages recovered by deduplication (see column *ZPG*) is low (78), the savings of the peak memory usage are assumed to be induced by the proactive avoidance of page duplicates via the optimized allocation and duplication strategies. For `GU/08a-2`, the optimization was not able to save memory for peak memory usage and no meaningful amount for the average memory usage was saved (*GainA*). The reason why `GU/08a-2` does not gain from the optimization is that even though it uses large vectors with 2.4 million elements, it allocates a vector that is immediately filled with random numbers. Thus, it does not profit from the optimized allocation and the content check can only recover a low number of zero pages as shown column *ZPG* (13). GU/08a-2 does not use any object duplication. Therefore, the optimized duplication has no potential for saving memory.

Even though the page-sharing optimization results in a slight increase of peak or average memory usage for the three benchmarks described above, all of the twelve other benchmarks benefit from the optimization with savings in both peak and aver-

age memory usage. We compute the geometric mean over all 15 benchmarks, thereby avoiding the impact of outliers on the geometric mean. The result is a reduction of peak memory usage by 13.6 % and a reduction of average memory usage by 18.0 %. Here, the highest amount of memory that could be saved occurs in the `lssvm` benchmark with 53.8 % for peak usage and in `randomForest` with 37.9 % for the average memory usage. Both of these benchmarks have a high number of zero pages recovered by the content check. Thus for those benchmarks, the reduction of the memory footprint is not just triggered by the allocation and duplication optimization but also by the dynamic refinement that deduplicates zero pages.

Table 7.3 shows summarized values for the memory consumption over the complete runtimes of all benchmarks. To gain additional insights into the memory consumption behavior, the complete profile of the memory usage over runtime will be also analyzed. The four most interesting memory consumption profiles for the benchmarks (`glmnet`, `gbm`, `randomForest`, and `naiveBayes`) are shown in Figure 7.7. For each benchmark, the run with the execution time closest to the average of its 10 executions is selected. The confidence intervals over all 10 runs of each benchmark are less than 1 %, thus the figure shows only the data from a single run. The x-axis represents the runtime in seconds while the y-axis represents the corresponding memory consumption of the benchmark. Both the profile for the standard R interpreter (yellow curves) and the interpreter including the page-sharing optimizations (green curves) are presented. The straight lines at the top indicate the peak memory usage, while the dotted lines mark the average memory usage.

glmnet As mentioned, the `glmnet` benchmark utilizes an already-existing memory optimization for sparse matrices. It is included in the evaluation to determine if the page-sharing optimization can offer additional memory savings in the presence of an optimization that applies specialized application knowledge. In the top left of Figure 7.7 the memory-over-time behavior of this benchmark is illustrated. While there is only a small improvement for the average memory usage (see dotted green line), 6.2 % of the peak memory consumption is saved (see lines on the top). The memory consumption curves show that at all local memory peaks the optimized version of the R interpreter saves a small amount of memory while the memory consumption during the remaining parts of the execution is largely unaffected. This results in only a minor reduction of the average memory consumption. Still, even in the presence of a very specific optimization for sparse matrices the page sharing optimization can offer additional memory savings. As can be seen from column *ZPG* (Table 7.3), savings are triggered by a large number of pages recovered by deduplication (46 877).

gbm Not all benchmarks benefit from the content checks, though. For example, Table 7.3 shows that in `gbm` only 464 zero pages are recovered. This benchmark benefits more from the optimizations in allocation and duplication. The corresponding memory-over-time behavior is shown in the top right of Figure 7.7. Here, the optimization does not reduce the peaks of the memory consumption, but there is a

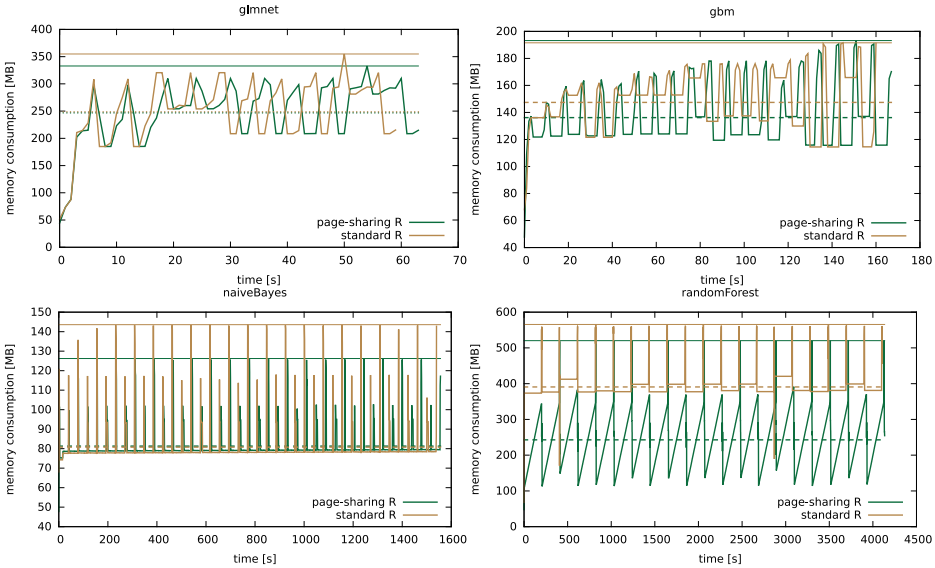


Fig. 7.7: Memory consumption over time profiles for benchmarks with different memory behavior for the standard R interpreter vs. the interpreter with the page-sharing optimization. Lines at the top indicate the peak memory usage; dotted lines mark the average memory usage [385].

marked reduction of memory usage in the valleys between the peaks, reducing the average memory consumption by 7.9 %.

naiveBayes Another benchmark that does not benefit from the content checks is `naiveBayes` with just 78 zeroed pages recovered. Its memory-over-time profile is illustrated in the bottom left of Figure 7.7. In `naiveBayes` only the peak memory usage is reduced by the optimization (large distance between the straight lines at the top), but the average memory usage (small distance between the dotted lines) is not affected. The profile also shows that `naiveBayes` has much smaller peaks compared with `gbm`. Thus, the large reduction of memory usage at those peaks results only in a small effect on the average memory consumption.

randomForest Finally, `randomForest` in the bottom right of Figure 7.7 represents one of the benchmarks where the recovery of zeroed pages triggers high memory savings. Here, the content checking reclaims 1 130 650 pages, which corresponds to slightly more than 4 GB of memory. The `randomForest` profile shows a saw-tooth curve for the optimized interpreter (see green curve). This indicates that the benchmark uses large blocks of memory that are slowly written to. For the page sharing optimizations, this represents an ideal memory usage pattern, as the allocation of memory is delayed until the benchmark writes data to it. This results in a 37.9 % improvement of the average memory consumption (large distance between dotted lines)—the average time during which the benchmark has a high memory consumption is thus reduced.

Looking back at the profile of `glmnet` (top left), the green curve that shows the profile for the optimized interpreter is longer than the yellow curve for the standard interpreter and there is an increasing shift between the peaks of both curves over time. The reason for this lies in the additional CPU time needed to provide the page-sharing optimizations. The runtime overhead induced by the memory optimization will be referred to in the next paragraph.

Runtime Overhead There are multiple reasons for the runtime overhead caused by the optimizations. For the 15 benchmarks shown so far, 4 have a runtime overhead of ≤ 1 %, an additional 6 have an overhead ≤ 5 %, an additional 2 have an overhead around 8 %, and the remaining 3 have an overhead between 13 % and 17 %. More details on the overhead are available in a separate publication [385].

Runtime Reduction In all previous measurements, the RAM available in the system was sufficient to hold all data used by the benchmark. If this is not the case, runtime overhead can become insignificant. This will be illustrated in the following. When the amount of RAM in the system is too small to hold all data required, there are situations where the proposed memory optimization is also able to reduce the runtime of the benchmark instead of adding overhead. This is due to frequent page swaps requiring I/O when the total capacity of RAM is exceeded, also known as "thrashing". To analyze this situation, two benchmarks are considered. The first one is the lssvm benchmark where the optimization provides a large reduction in memory consumption. The second benchmark is an instance of logreg where the optimization provides only smaller memory gains.

For the analysis, the memory requirements of the benchmarks need to be increased beyond the capacity of the RAM in the system. Instead of increasing the dataset size of both benchmarks, the system is limited to just 1 GB of RAM, since the runtimes of the benchmarks do not scale linearly with the dataset size, leading to excessively high execution times. However, since the logreg benchmark has a much smaller memory consumption than 1 GB, the dataset size for logreg is increased to 70 000 samples with 300 numeric features. This increases the memory requirements of this benchmark to approximately the same level as lssvm. This still results in acceptable execution times for logreg.

Table 7.4 shows the results for the previous 6 GB system configuration and the limited 1 GB RAM configuration for both benchmarks. The logreg benchmark is now shown as logreg-2 because it was executed with the previously described larger dataset. In the 1 GB configuration, the system had to swap for both the standard and optimized interpreters, resulting in a large increase in runtime compared with the 6 GB configuration. The peak memory usage for the interpreters is identical in both configurations while the average memory usage differs because this value is time-dependent and thus influenced by swapping. This swapping also increases the variability in the runtime

Tab. 7.4: Evaluation results with two configurations of RAM; Std – standard R interpreter; Opt – optimized R interpreter; Gain – relative gain; Speedup: runtime speedup factor (Std / Opt). Confidence intervals (C) for runtime are shown; others are ≤ 0.8 % [385].

Benchmark	Std Peak [MB]	Opt Peak [MB]	Gain Peak [%]	Std Avg [MB]	Opt Avg [MB]	Gain Avg [%]
logreg-2, 1 GB	1228.2	1094.8	10.9	965.7	789.6	18.2
logreg-2, 6 GB	1228.2	1094.8	10.9	967.8	823.2	14.9
lssvm, 1 GB	1365.1	631.1	53.8	970.0	381.3	60.7
lssvm, 6 GB	1365.1	631.0	53.8	820.2	381.1	53.5

Benchmark	Std [s]	Opt [s]	Speedup (CI)
logreg-2 1 GB	6395.5	5785.6	$1.105^{1.144}_{1.071}$
logreg-2, 6 GB	579.8	598.5	$0.969^{0.971}_{0.967}$
lssvm, 1 GB	3080.3	593.8	$5.188^{5.350}_{5.029}$
lssvm, 6 GB	530.5	601.2	$0.882^{0.885}_{0.880}$

measurements, thus the confidence intervals for the speedup factors are also included (see lower part of Table 7.4).

Reducing the available memory from 6 GB to 1 GB drastically increases the runtime for both versions, the standard R interpreter (*Std*) and the interpreter including the memory optimization (*Opt*). Still, the reduction in memory consumption for logreg-2 has turned the slowdown (factor 0.969) in its 6 GB configuration into a small speedup (factor 1.105) when the RAM is limited to 1 GB. Depending on the benchmark and its memory usage pattern, a different situation could also happen. In the worst case, the content check of the optimized interpreter touches a large number of pages, forcing them to be swapped in. This additional swap activity can increase the runtime so that the gains from a reduced memory footprint may become irrelevant. The second benchmark `lssvm` shows something closer to the best case for the optimization: Here, the page-sharing optimization manages to save enough memory to avoid swapping. In this case, significant speedups are gained, as shown in the lower part of Table 7.4 for the 1 GB configuration of `lssvm`.

Similar to `logreg-2`, memory usage does not vary much between both configurations (see upper part of Table 7.4). Considering the runtime results, the optimized interpreter *Opt* only needs 593.8 seconds to run the `lssvm` benchmark. This is almost unchanged from the 6 GB configuration (601.2 seconds). By contrast, the standard interpreter *Std* has now increased its runtime to 3080.3 seconds (51.3 min.) when limited to 1 GB of RAM. This makes the overhead of the memory optimization irrelevant because the time gained by avoiding page I/Os is much larger. The page-sharing optimization enables a speed up by a factor of 5.2 for llsvm by reducing the peak memory consumption by 53.8 %. This speed up is also illustrated in Figure 7.8. It shows the

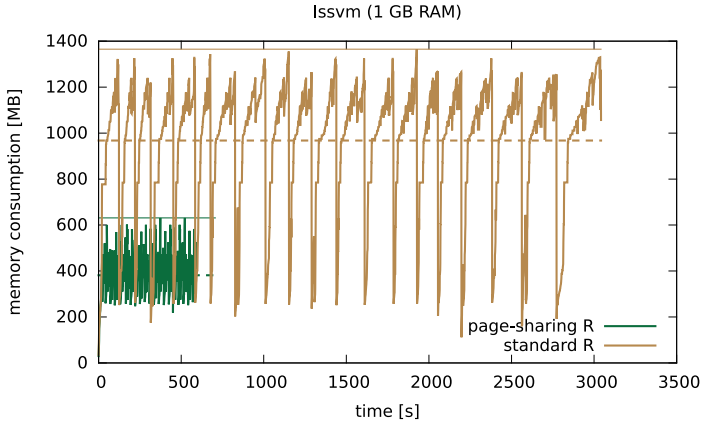


Fig. 7.8: Memory consumption over time profile for the lssvm benchmark. Speed-up reaches a factor of 5.2 on a system with 1GB of RAM. Solid lines indicate the peak memory and dotted lines mark the average memory usage [385].

memory consumption profile for one exemplary execution of the `lssvm` benchmark. This demonstrates that reducing the memory consumption with the page-sharing optimization can significantly improve the runtime for memory-hungry benchmarks if the available RAM is constrained. In turn, this can enable the processing of larger datasets.

7.1.5 Summary

The R interpreter induces a large memory overhead in the machine learning applications, due to wasteful memory allocation [387]. The goal of the presented memory optimizations was to enable efficient memory utilization, especially for memory-hungry R applications like machine learning algorithms. To accomplish this goal, this contribution presented an application-transparent memory optimization employing page sharing at a memory management layer between the R interpreter and the operating system's memory management. The optimization benefits a large number of applications since it preserves compatibility with the available software libraries that most R programs are based on, and covers one of the most important resource bottlenecks of machine learning algorithms. By concentrating on the most rewarding optimizations—the sharing of zero-filled pages and deduplicating at the page level instead of the object level—the overhead of more general OS level memory optimization approaches such as deduplication and compression is avoided. With the proposed optimization, considerable reductions of the memory consumption for a large number of typical real-world benchmarks have been achieved. This is an important step towards processing larger input sizes. It also significantly speeds up the computation in cases where previously pages had to be swapped out due to insufficient main memory.

7.1.6 Conclusion

Designers of machine learning applications should be allowed to focus on the functionality of their algorithms. In order to execute these on resource-constrained embedded systems, possible optimizations of the implementation should be performed. The presented work demonstrates the benefits of such optimizations for the case of memory resources. In addition to the other optimizations in this contribution, we conjecture that more memory-oriented optimizations exist and propose that they should be exploited in order to execute machine learning algorithms in particular on hardware with limited amounts of memory.

7.2 Machine Learning Based on Emerging Memories

Mikail Yayla
Sebastian Buschjäger
Hussam Amrouch

Abstract: Due to the exceptional recent developments in deep learning, many fields have benefited from the application of Artificial Neural Networks (ANNs). One of the biggest challenges in ANNs, however, is the resource demand. To achieve high accuracy, ANNs rely on deep architectures and a massive amount of parameters. Due to this, the memory sub-system is one of the most significant bottlenecks in ANNs.

To overcome the memory bottleneck, recent studies have proposed using approximate memory in which the supply voltage and access latency parameters are tuned for lower energy consumption and for faster access times. However, these approximate memories frequently exhibit bit errors during the read process. Typical software solutions that monitor and correct these errors require a large processing overhead that can negate the performance gains of executing ANNs on these devices. Hence, error-tolerant ANNs that work well under uncorrected errors are required to prevent performance degradation in terms of accuracy and processing speed.

In this contribution, we review the available and emerging memories that can be used with ANNs, with a focus on approximate memories, and then present methods to optimize ANNs for error tolerance. For memories, we survey existing memory technologies such as Static Random-Access Memory (SRAM) and Dynamic Random Access Memory (DRAM), but also present emerging memory technologies such as Ferroelectric FET (FeFET), and explain how the modeling on the device level needs to be performed for error tolerance evaluations with ANNs. Since most approximate memories have similar error models, we assume a general error model and use it for the optimization and evaluation of the error tolerance in ANNs. We use a novel hinge loss based on margins in ANNs for error tolerance optimization and compare it with the traditional flip regularization. We focus on Binarized Neural Networks (BNNs), which are one of the most resource-efficient variants of ANNs.

7.2.1 Introduction

Artificial neural networks have been applied successfully in numerous fields, and are being executed on a variety of systems ranging from large computing clusters to small, battery-driven embedded systems. In most cases, state-of-the-art neural network models rely on a large number of parameters to achieve high performance. This leads to an expensive, slow, and energy-consuming *memory bottleneck*. On neural network acceler-

atorswith SRAM, the energy consumption of the memory makes up the largest fraction of system energy, while advances in memory bandwidth are significantly slower than processing speed. Hence, improving the memory consumption of ANNs and improving the memory sub-systems is imperative to further push the applications of ANNs. One design paradigm to improve the memory sub-system is to use approximate memory in which resource efficiency is achieved by allowing for bit errors during the read and/or write process. Likewise, reducing the memory consumption of ANNs is an established part of deep learning research. Here, arguably, the most extreme form is to use Binarized Neural Networks (BNNs) that only use binary weights $\{0, 1\}$ leading to a potential 32 times memory reduction as high as 32 times that of their floating-point siblings. Interestingly, it has been shown that BNNs can be trained to tolerate bit errors by bit flip injections during training. However, this method has a large overhead and does not scale well with model size and higher bit error rates .

In this contribution, we first summarize the currently available and emerging memories that are possible to be used with neural network inference systems. Here, we focus on approximate memories, which are unreliable due to bit errors and for which countermeasures are necessary. One of the most promising emerging memory components is the FeFET, which has high speed, and low energy consumption, but faces reliability issues. We explain how FeFET can be used as approximate memory for neural networks despite the bit errors caused by temperature and read voltage. Finally, we present results on how bit error tolerance in ANNs is achieved without bit flip injections based on margin-maximization and compare it to the traditional methods for bit error tolerance optimization of ANNs. This contribution was previously published as a conference paper in [113].

7.2.2 Emerging Memories

Recent studies on efficient ANN-based inference systems have explored the use of approximate memory, which has been realized by reducing the memory supply voltage and tuning latency parameters with the goal of lower power consumption and faster access. If these methods are pushed to the limit, high Bit Error Rates (BERs) can occur. Before discussing bit errors and how to deal with them in more detail we will quickly survey volatile memories (SRAM, DRAM) and other emerging non-volatile memories (FeFET, Resistive Random Access Memory (RRAM), Spin Transfer Torque Random Access Memory (STT-RAM) or Magnetoresistive Random Access Memory (MRAM)) here.

SRAM For ANN inference systems using on-chip SRAM, the works in the literature mainly employ scaling of various device parameters. To reduce energy consumption, the SRAM voltage is scaled in [306, 652]. Yang et al. [717] separately tune the weight and activation values of BNNs to achieve fine-grained control over the energy consumption. Sun et al. [652] propose similar techniques for ternary ANNs. A similar approach is

employed by Henwood et al. [306], in which layer-wise the best energy-accuracy trade-off for SRAM is chosen.

DRAM For DRAM, the study by Koppula et al. [381] provides an overview of studies related to ANNs that use different DRAM technologies and proposes a framework for evaluating ANN accuracy when using approximate DRAM in various different settings and inference systems. Specifically, the study shows that DRAM parameters can be tuned such that energy and performance are optimized to achieve significant improvements, whereas the ANN accuracy drop stays negligible due to the ANNs' adaptations in retraining. Other studies, e.g. [532, 672], also optimize the refresh rate of DRAM to achieve energy savings.

RRAM Hirtzlin et al. [316] propose computing BNN operations with RRAM that features in-memory processing capabilities. They set the write energy of RRAM low and show that BNNs can tolerate the resulting errors by error tolerance training. This low-energy setting also increases the RRAM cell lifetime since low-energy writes stress the cells less. The work by Yu et al. [727] also uses RRAM to implement on-chip BNNs. They show that under limited bit yield, BNNs can still operate with satisfying accuracy. Sun et al. [651] propose an RRAM synaptic array to deploy BNNs. They investigate the accuracy impact of errors from sense amplifiers that have offsets due to process variation.

MRAM or STT-RAM Another branch in the literature is about ANNs on STT-RAM or MRAM. Hirtzlin et al. [315] propose deploying BNNs on MRAM with a low-energy programming setting that causes relatively low error rates, and no significant accuracy drop, but decreases write energy by a factor of two. Tzoufras et al. [675] also propose operating BNNs on MRAM with reduced voltage with similar results. They test a wide range of error rates and discuss the implications of BNN bit error tolerance on the lifetime, performance, and density of MRAM. Pan et al. [549] take a different approach for energy reduction and investigate the benefits of multi-level cell MRAM for the in-memory acceleration of BNNs. For more general ANN models, Vincent et al. [686] propose tunable STT-RAM to save resources.

FeFET FeFET is considered to be one of the most promising memory technologies. The reason why FeFET store logic '0' and logic '1' lies in the available dipoles inside the FE. The directions of these dipoles can switch if a sufficiently strong electric field is applied. This state is non-volatile because the dipoles retain their direction when the field is turned off. The logic '0' and logic '1' can be read out from the FeFET based on the intensity of the current returned (e.g. high or low), which can be converted into the digital domain with sensing circuits.
The three main advantages of FeFET over other non-volatile memories are as follows:

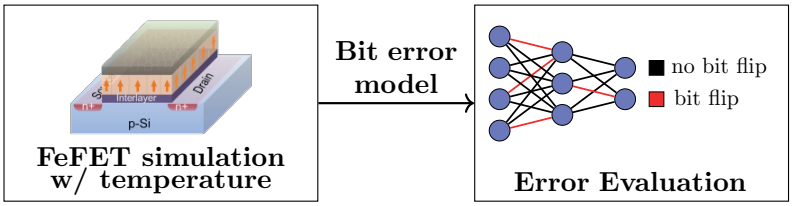


Fig. 7.9: Errors due to temperature, stemming from underlying FeFET devices, are modeled and then injected during the ANN inference [720].

1. FeFET is fully CMOS-compatible, which means that it can be fabricated using current manufacturing processes. This has been demonstrated by GlobalFoundries [668].
2. FeFET-based memories can perform read operations within 1ns latency. This reduces the differences to traditional SRAM technology, while the energy usage of FeFET is significantly lower [668].
3. FeFET memory has the potential to enable extremely low-density memory since the core cell consists merely of a single transistor.

One of the major disadvantages of FeFETs is error susceptibility. Manufacturing variability (i.e. process variation during production) and temperature fluctuations at run-time can cause variations in the FeFET properties. This shrinks available noise margins and may cause errors. To still employ FeFETs despite the errors in, say, on-chip memory for Binarized Neural Networks (BNNs) inference systems, it is necessary to extract the error models for the stored bits. With the error model, the impact of the temperature-induced bit errors on the inference accuracy of BNNs can be evaluated.

In Figure 7.9, the steps for extracting the temperature-dependent error model of FeFET transistors are shown. The entire FeFET device has been implemented and modeled in the Technology CAD (TCAD) framework (Synopsys Sentaurus [656]). The variation in the underlying transistor and the added ferroelectric layer are considered. After incorporating the temperature and variation effects in the calibrated TCAD models, Monte-Carlo simulations for the entire FeFET device are performed. Then the probability of error is extracted for a certain read voltage, i.e. the probability that logic '0' is read as logic '1' and a logic '1' is read as logic '0'. Details on device physics modeling and reliability analysis for FeFET under the effects of temperature variability (runtime) and manufacturing (design-time) variability can be found in [280] and [534], respectively.

7.2.3 Binarized Neural Networks

Traditional neural networks use floating-point (e.g., 32 bits) or integer values (e.g., 8 bits) to represent the ANN parameters (i.e., weights, activations, inputs, etc.) . In such

a case, the position of the occurred bit error (i.e., the bit flip in the value) does matter. Specifically, in floating-point ANNs, a one-bit error in one weight can cause the prediction of the ANN to become useless (see e.g. [381]). This typically occurs when a bit flip in the exponent of the floating-point representation occurs leading to an error with an unacceptable magnitude. As mentioned before, BNNs are resource-efficient neural networks that are ideally suited for small devices. Additionally, they can be trained to be resilient against bit errors, which makes them ideal candidates for approximate memories. In BNNs each weight (and possibly each activation) is stored in a single bit $\{0, 1\}$. Hence, a bit error in a binary weight or binary input causes a change of the computation result by merely 1, reducing its overall impact. In addition to the reduced impact of bit errors and reduced memory footprint due to smaller weights the execution of BNNs also becomes simpler. Consider, for example, the output of the fully connected l-layer with activation σ and weights W^l

$$f^l(X) = \sigma(W^l X) \tag{7.1}$$

In regular floating-point neural networks, the execution of this layer requires the repeated computation of matrix-vector products $W^l X$ as well as the application of σ. In a BNN this operation becomes

$$2\texttt{popcount}(\texttt{XNOR}(W^l, X)) - B > T \tag{7.2}$$

where popcount counts the number of 1s in the XNOR-result, B is the number of bits in the XNOR operands, and T is a learnable threshold parameter if batch normalization layers are used, whose comparison produces binary values (representing a shifted binarization function) [325, 609].

A common method of training ANNs is to apply stochastic gradient descent (SGD) with mini-batches. Let $\mathcal{D} = \{(x_1, y_1), \ldots, (x_I, y_I)\}$ be the training data with $x_i \in \mathcal{X}$ as the inputs, $y_i \in \mathcal{Y}$ as the labels, and $\ell\colon \mathcal{Y}\times\mathcal{Y} \to \mathbb{R}$ as the loss function. $W = (W^1, \ldots, W^L)$ are the weight tensors of layer $1 \ldots L$ and $f_W(x)$ is the output of the ANN. The goal is to find a solution for the optimization problem

$$\arg\min_W \frac{1}{I} \sum_{(x,y)\in\mathcal{D}} \ell(f_W(x), y) \tag{7.3}$$

with a mini-batch SGD strategy that computes gradients using backpropagation.

To train BNNs, Hubara et al. [325] proposes to deterministically binarize the weights and activations during the forward pass. For backpropagation, the floating-point numbers are used for parameter updates. This leads to training times similar regular ANNs but assumes binary values during the forward pass. More formally, let $b\colon \mathbb{R} \to \{-1, +1\}$ be a binarization function with

$$b(x) = \begin{cases} 1 & x > 0 \\ -1 & \text{else} \end{cases} \tag{7.4}$$

and let $B(W)$ denote the element-wise application of b to a tensor W. Now we simply apply B during the forward pass to each weight tensor. During the backward pass, the authors propose using full floating point precision, whereas during the backward-pass they replace the gradient of b with the straight-through estimator. Consider the forward computation $Y = B(X)$. Let $\nabla_Y \ell$ denote the gradient with respect to Y. The straight-through estimator approximates

$$\nabla_X \ell := \nabla_Y \ell, \tag{7.5}$$

essentially pretending that B is the identity function. Algorithm 5 summarizes this approach.

Algorithm 5: Binarized forward pass for a network with L layers, each with weight tensors W^l performing a generic operation $\circ^l$ (e.g. a convolution).

```
for l ∈ {1, ..., L} do
    x ← B(B(W^l) ∘^l x)
```

7.2.3.1 Flip Regularization

To make BNNs bit error-tolerant, the state-of-the-art method is bit flip injections in the binarized values during the forward pass, as proposed by Hirtzlin et al. [316]. The idea is simple: To make BNNs robust against bit errors, we simulate the errors already during training time. During each forward pass computation, we generate a random bit-flip mask and apply it to the binary weights.

Let M denote a random bit-flip mask with entries ± 1 of the same size as W that we multiply component-wise to the binarized weights. We first consider computing the bit-flip operation as $H = (B(W) \cdot M) \circ X$ where $\circ$ denotes the application of the ANN to the input X. Standard backpropagation on a loss ℓ that is a function of H yields the following gradient of ℓ with respect to $B(W)$

$$\nabla_{B(W)} \ell = M \cdot \nabla_{B(W) \cdot M} \ell \tag{7.6}$$

which for fully connected layers amounts to a gradient update

$$\nabla_{B(W)} \ell = M \cdot (\nabla_H \ell \, X^T). \tag{7.7}$$

We see that an update computed this way accounts for the bit-flips that were performed. We propose instead using a special flip-operator with straight-through gradient approximation. We denote by e_p the bit error function that flips its input with probability p and let E_p denote its component-wise version. During training, we change the forward pass such that it computes

$$X^{l+1} := B(E_p(B(W^l)) \circ X^l). \tag{7.8}$$

We replace the gradient of E_p with a straight-through approximation. This way, in the example above we now have $H = E_p(B(W)) \circ X$ with gradient updates $\nabla_{B(W)}\ell = \nabla_{E_p(B(W))}\ell$ which for fully connected layers yields the update

$$\nabla_{B(W)}\ell = \nabla_H \ell \, X^T \tag{7.9}$$

which is unaware of bit flips and just uses the corrupted outputs H.

The original bit-flip regularization proposed in [316] reports extreme overfitting to the flip probability used during training. As we will see later in the experiments, we do not report such an overfitting. We believe that the approach using straight-through gradient approximation is superior and that the extreme overfitting is attributable to the use of the naive gradient.

7.2.3.2 Margin-Maximization for Bit Error Tolerance Optimization

Bit-flip regularization improves the error tolerance of the network by simulating bit errors during the forward pass. This introduces two objectives to the training: Given a set of labeled input data, train a BNN for high accuracy and for high bit error tolerance. Hence, another approach is to combine high accuracy and high bit error tolerance into a single loss function directly so that both objectives are jointly optimized during training. To do so, we now introduce a margin-based neuron-level bit error tolerance metric for BNNs that is then extended to formulate a bit error tolerance metric for the output layer.

In the following, we use a notation describing the properties of neurons in convolutional layers, but our considerations also apply to neurons in fully connected layers. Let n be the index of one neuron in a ANN, and $x \in \mathcal{X}$ an input to the ANN. The output of a neuron in a convolutional layer is a feature map with height U and width V. Let $h_{x,n,u,v} \in \mathbb{Z}$ be the pre-activation value of neuron n at place $(u, v) \in \{0, \ldots, U\} \times \{0, \ldots, V\}$, *before* applying the activation function. For BNNs, the pre-activation values of a neuron are computed by a weighted sum of inputs and weights that are ±1. Therefore, one bit flip in one weight changes the pre-activation value by 2.

Theorem 25. *Let $n \in \{0, \ldots, N\}$ be the index of one neuron. Furthermore, let q be the number of bit flips induced in the weights of neuron n. The pre-activation of neuron n at place (u, v) after induction of these bit flips is in the interval $[h_{x,n,u,v} - 2q, h_{x,n,u,v} + 2q]$.*

The proof can be found in [113].

A detailed analysis of the error tolerance for hidden-layer neurons has been conducted in [114], but the use of Theorem 25 for optimizing bit error tolerance on the neuron-level has been reported to be unsuccessful. We hypothesize that bit flips of neuron outputs can only affect the BNN prediction if the effect of bit flips reaches the output layer and leads to a change in the predicted class. Therefore, we now shift our focus on applying the notion of margin to the output layer, i.e., to neurons with index in N_O.

Each neuron in the output layer has only one output value $h_{x,n,1,1}$ which is one entry in the vector of predictions $\hat{y}$. No activation function is applied to the output value of these neurons. There are as many values in $\hat{y}$ as there are neurons in the last layer. The index of the entry with the maximum value in $\hat{y}$ determines the class prediction, where we assume that ties are broken arbitrarily.

If bit errors modify the output values in the output layer such that another neuron provides the highest output value, then the class prediction changes. Let $h_{x,n',1,1}$ and $h_{x,n'',1,1}$ with $n', n'' \in N_O$ be the highest and the second-highest output value of neurons in the output layer. The following corollary shows that the margin

$$m := h_{x,n',1,1} - h_{x,n'',1,1} \tag{7.10}$$

serves as a bit error tolerance metric for the output layer.

Corollary 26. *If $m > 0$, then the output layer of the BNN tolerates* $\max(0, \lfloor \frac{m}{2} \rfloor - 1)$ *bit flips.*

The proof can be found in [113].

We now focus on constructing a loss function based on Corollary 26 and the hinge loss known from Support Vector Machines (SVMs). The hinge loss [602] for maximum margin classification is defined as

$$\ell(y, f) = \max(0, 1 - y \cdot f), \tag{7.11}$$

with the ground truth prediction $y = \pm 1$ and the prediction $f \in \mathbb{R}$. This loss becomes small if the predictions have the same sign as the predicted class and are close to 1 in magnitude. For predicted values larger than 1, the loss becomes 0. The "1" in the loss forces the classifier to maximize the margin between two class predictions.

For BER tolerance of the last layer, the margin m as introduced in Equation 7.10 needs to be large so that the maximum number of bit flips the output layer can tolerate is high. The margin can be directly computed by subtracting the second-highest entry $\hat{y}_{c''}$ of the output vector $\hat{y}$ from the highest entry $\hat{y}_{c'}$, i.e., $m = \hat{y}_{c'} - \hat{y}_{c''}$. However, optimizing with respect to m without considering the other entries $\hat{y}_c$ of $\hat{y}$ may not exhaust the full potential of the margin between $\hat{y}_{c'}$ and the output of the other classes $\hat{y}_c$. The larger the margin between $\hat{y}_{c'}$ and $\hat{y}_c$ of other classes c, i.e. $m_c = \hat{y}_{c'} - \hat{y}_c$, the more bit errors can be tolerated in the neuron that calculates $\hat{y}_c$ without a change in the prediction. To put it concisely, for a bit error tolerant output layer, $\hat{y}_{c'}$ needs to be as large as possible, while the other $\hat{y}_c$ need to be as small as possible.

In the case of BNNs for multi-class problems, however, the version of the hinge loss in Equation 7.11 cannot be directly used. To extend the hinge loss to multiple classes, we define y_{enc} as a one-hot vector with elements in $\{-1, 1\}$, which has a +1 at the index with the ground truth, else −1. y_{enc} has the same number of elements as $\hat{y}$. Then the element-wise product $y_{enc} \cdot \hat{y}$ is computed. In this product, in case of correct predictions, positive predictions in the correct class will stay positive, and negative predictions that

Tab. 7.5: Datasets used for experiments.

Name	# Train	# Test	# Dim	# classes
FashionMNIST	60000	10000	(1,28,28)	10
CIFAR10	50000	10000	(3,32,32)	10

Tab. 7.6: Parameters used for experiments.

Parameter	Range
Fashion FCNN	In → FC 2048 → FC 2048 → 10
Fashion CNN	In → C64 → MP 2 → C64 → MP 2 → FC2048 → 10
CIFAR10 CNN	In → C128 → C128 → MP 2 → C256 → C256 → MP 2 → C256 → C256 → MP 2 → FC 2048 → 10

should be as negative as possible become positive. In case of wrong predictions, i.e. high negative values for the correct class and high positive values for the wrong class, the values become negative. For a high penalty in the wrong case and a small penalty for the correct case, we subtract the product $y_{enc} \cdot \hat{y}$ from a parameter b, and get $(b - y_{enc} \cdot \hat{y})$. Since we do not demand higher prediction values than b, we set negative values to zero with the max function, and denote the Modified Hinge Loss (MHL):

$$\ell_{MHL}(\hat{y}, y_{enc}) = max\{0, (b - y_{enc} \cdot \hat{y})\}. \tag{7.12}$$

7.2.4 Experiments

We evaluate fully connected binarized neural networks (FCBNNs) and convolutional binarized neural networks (CBNNs) in the configurations shown in Table 7.6 for the datasets FashionMNIST and CIFAR10 (see Table 7.5). In all experiments, we run the Adam optimizer for 100 epochs for FashionMNIST and 250 epochs for CIFAR10. We use a batch size of 128 and an initial learning rate of 10^{-3}. To stabilize training, we exponentially decrease the learning rate every 25 epochs by 50 %. In the following, we compare the margin-based methods (MHL) to Flip Regularization (FR). FR uses the Cross-Entropy Loss (CEL) by default. We first compare MHL without FR to FR. In a second step, we compare MHL without FR to MHL in combination with FR.

7.2.4.1 MHL Only vs. FR

Figure 7.10 presents the experimental results of different BNNs with respect to the accuracy over BER (from 0 % to up to 15 % in Figure 7.10(a) and (b), and from 0 % to up

to 5 % in Figure(c)). For each dataset, five BNNs were conducted using MHL without any FR and FR with different BERs for bit-flip injections. Moreover, for all BNNs trained with MHL, we employed a parameter search for b, testing powers of two, up to two times the maximum value the neurons in the output layer can compute (maximum output value of a neuron in the output layer is the number of neurons in the layer before the output layer). Among these configurations of b, the best one was chosen. We observe that BNNs trained with the MHL without FR have better accuracy over BER than the BNNs trained with FR, i.e., in Figure(a) and (b) up to 10 %, and in Figure(c) up to 5 %. The BNNs trained with FR suffer from a significant accuracy drop for lower BERs, when the BER during training is high, e.g., CEL 20 % and/or CEL 30 % at low BER. The BNNs trained with MHL, however, do not suffer from this accuracy drop. Although the BNNs trained with FR 20 % and bit-flip injections have better accuracy for Fashion CBNN in Figure 7.10(b) when the error rate is higher than 10 %, the accuracy of the BNNs drops by a significant amount, which may be unacceptable. Below, we thus present further investigations.

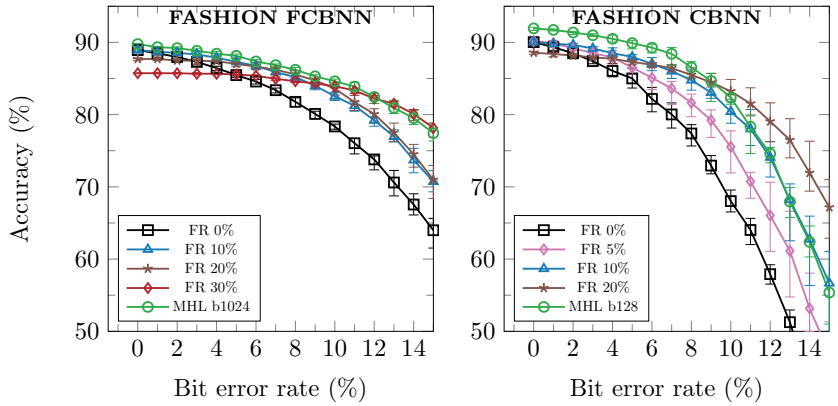
7.2.4.2 MHL Combined With FR

We evaluate BNNs trained with the MHL and FR under different BERs. In addition, the BNNs trained with the MHL without FR (i.e., those BNNs generated using the MHL in Figure 7.11 under 0 % BER) are included here as the baseline in this subsection. For all configurations, we employed the same parameter search for b as in the previous section. Figure 7.11 presents the experimental results of different BNNs with respect to the accuracy over BER (from 0 % to up to 30 % in Figure 7.11(a) and (b) and from 0 % to up to 6 % in (c)). In all experiments, we observe that the accuracy over the BER of the BNNs trained under MHL and FR is significantly higher than that of the baseline trained by only MHL. For example, for Fashion in Figure 7.11, the BER at which the accuracy degrades significantly is extended from 5 % (baseline, green curve) to 20 % and 15 %, respectively, with a small trade-off in the accuracy at 0 % BER. If more accuracy at low error bit rates is traded, the BER at which accuracy degrades steeply can be shifted even further. For CIFAR10 in Figure 7.11, this breaking point can also be increased. However, more accuracy has to be traded compared with previous cases. If b is higher than the ones shown, the accuracy for lower BERs suffers similarly to how it would using CEL with high BERs. If b is lower, there will be no significant change compared with CEL with 0 % BER. We only show the results with the best b.

7.2.5 Conclusion

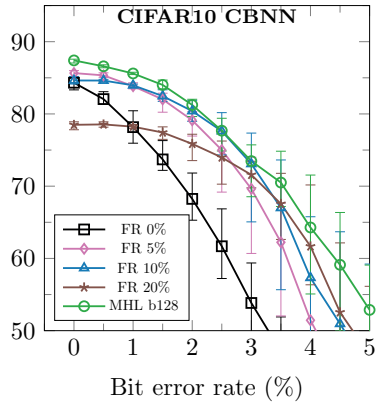
Deep learning is notoriously memory hungry and hence new memory sub-systems must be developed to push the application of ANNs to small devices. Likewise, new ANN architectures can help to reduce memory consumption and offer a more resource-

friendly execution of deep networks. Non-volatile memories such as Ferroelectric FET (FeFET) are a promising technology for new memory sub-systems. FeFET enables faster and more energy-efficient read/write operations but it introduces bit errors into the execution. While standard software solutions can monitor and correct bit errors, they negate the advantages of non-volatile memories by introducing further processing overhead. Neural networks that are resilient to random bit errors by design, on the other hand, can retain the advantages of non-volatile memories leading to potentially faster and more energy-efficient solutions. BNNs are a novel class of small, resource-efficient neural nets that are ideally suited for such a setting. In BNNs each weight consists of weights $\{0, 1\}$ so that they require 32 times less memory than their floating-point counterpart while being more resilient to random bit flips. In this contribution, we provided an in-depth discussion of the bit errors in BNNs and derived a novel max-margin optimization from it. Our approach offers a better accuracy across most error rates while preventing the overfitting of the BNN to a specific error rate. Hence, our approach allows the deployment of BNNs on a variety of different devices with unknown and varying error rates.



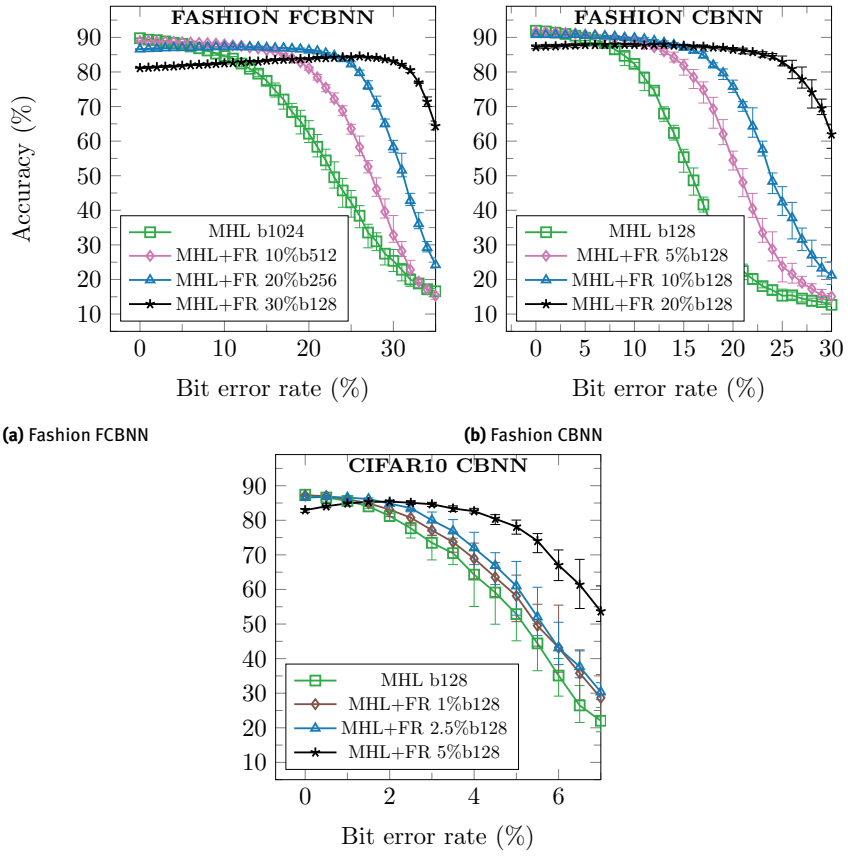
(a) Fashion FCBNN

(b) Fashion CBNN



(c) CIFAR10 CBNN

Fig. 7.10: Accuracy over bit error rate for BNNs trained with FR under a given bit flip injection rate (specified in the legend, 0 %, 5 %, 10 %, etc.) and BNNs trained with MHL without FR for a specified b in Equation 7.12.



(a) Fashion FCBNN

(b) Fashion CBNN

(c) CIFAR10 CBNN

Fig. 7.11: Accuracy over bit error rate for BNNs trained with MHL and FR (denoted as FR 0 %, 1 %, etc). The number after the *b* is the value to which the parameter *b* in the MHL is set during training (see Equation (7.12)).

7.3 Cache-Friendly Execution of Tree Ensembles

Sebastian Buschjäger
Kuan-Hsun Chen

Abstract: Ensembles of decision trees are among the most used classifiers in machine learning and regularly achieve state-of-the-art performance in many real-world applications, e.g., in the classification of celestial objects in astrophysics, pedestrian detection, etc. Machine learning practitioners are often concerned with model training, re-training different models again and again to achieve the best performance. Nevertheless, once a learned model is trained and validated, the executing cost of its continuous application might become the major concern.

Applying decision trees for inferences is very efficient in run-time, but it requires many memory accesses to retrieve nodes. For example, it is common to train several thousand trees, e.g., each with depth 15 leading to $2^{15} = 32\,768$ nodes per tree. This leads to millions of decision nodes that must be stored in memory and processed. Cache memory is commonly adopted to hide the long latency between the main memory and the processor. However, an improper memory layout might bring up additional cache misses, leading to performance degradation. Thus, designing a suitable memory layout of tree ensembles is of key importance to achieve efficient inference over tree ensembles.

In this contribution, we discuss the deployment of tree ensembles on different hardware architectures. Given a pre-trained decision tree ensemble, we first present different realization techniques commonly used in the literature. Afterwards, we study different layout strategies to optimize the node placement in the memory, focusing on the caches available on different hardware architectures. Finally, we present the evaluation results over different configurations and combine all approaches into a single framework that automatically generates the optimized realization for a target hardware architecture.

7.3.1 Introduction

Efficient learning has always been the focus of research, but the demand for the *efficient application* of learned models has emerged only recently. Consider, for example, self-driving cars. Current prototypes use machine learning (ML) for image recognition and fundamental steering.[1] Thus, the ML model must not only be applied continuously, but it also must react on time. As a second example, consider search engines that utilize ML

1 ,https://towardsdatascience.com/teslas-deep-learning-at-scale-7eed85b235d3.

models such as Gradient Boosted Trees[2] to rank search results. These engines routinely process roughly 12 billion search queries a month worldwide.[3] The 4 480 287 queries per second they process demand fast model application.

While deep learning is excellent for unstructured image data, tree ensembles are often referred to as one of the best black-box methods available for structured data. They offer high accuracy with only a few parameters to tune [120, 223] and frequently place among the top methods in data science competitions.[4] For real-time application, tree ensembles have become important in many domains, e.g., the real-time classification of celestial objects in astrophysics [115], real-time pedestrian detection [466], real-time 3D face analysis [211]), the real-time classification of noise signals [608], nano-particle sensors [439].

However, these trees are usually stored in the main memory and processed directly out of the memory. The runtime of such a memory-intensive application is mainly determined by the use of the various caches of the CPU. Surprisingly, as the line between realizational details and algorithmic contributions becomes blurry on modern computing systems, caching behavior determines the performance of implemented algorithms even more than algorithmic differences [615]. For tree ensembles, we can foresee that an analytical approach to an efficient layout of the memory is desirable. Given a pre-trained tree ensemble, we present several cache-aware approaches to optimize the memory layout (so-called tree-framing), while preserving the original ensembles' accuracy. The proposed approaches are wrapped in a code generator that automatically adapts to underlying architectures to produce optimized code segments. Overall, we present the following contributions:

Cache-aware tree-framing approaches We analyze the source of cache misses on two common tree realizations, i.e., *native* and *if-else* trees, and discuss several approaches at the application level to optimize the memory layout by artificially creating instruction/data locality .

Architecture-aware code generator We present a code generator that exploits the analytical insights for generating optimized realizations of a given tree ensemble. The code generator is publicly available at https://github.com/sbuschjaeger/fastinference.

Empirical evaluation We perform 1800 experiments across three different computer architectures and show that our approaches offer a speed-up factor at $2 - 4$ on average without changing the prediction accuracy of the given trained model.

This contribution was previously published as a conference paper in [108] and was later expanded in a dissertation in [107].

2 https://www.seroundtable.com/bing-core-ranking-algorithm-machine-learning-27040.html.

3 Numbers are for 2019, see https://www.statista.com/topics/4294/bing/.

4 https://www.kdnuggets.com/2016/01/anthony-goldbloom-secret-winning-kaggle-competitions.html.

7.3.2 Related Work

Tree ensembles are some of the most used machine learning algorithms and, as such, have been studied extensively in the literature. In the context of model application and fast inference, there are two principled approaches. The first set of methods changes the training procedure for Decision tree (DT) ensembles to produce more resource-friendly models. This can be beneficial to achieve the highest accuracy given the computational resources provided, but often result in longer training times and more evolved training procedures. Common examples for this approach are pre- and post-pruning rules for trees (see, e.g. [43]) or the pruning of entire ensemble members [347, 449, 589, 739].

The second set of methods studies the realization of a given DT ensemble and its execution. This approach uses the ensemble as-is and, as such, does not affect the training. We will focus on this methodology in this contribution. Note that both methods can also be combined. For example, Van Essen et al. present in [679] a comprehensive study of different architectures for implementing Random Forests (RFs) on CPUs, FPGAs, and GPUs. Based on the CATE algorithm [586], the authors train an RF with DTs constrained by a fixed height. By fixing the tree-depth, the authors show a practical pipelining approach for executing DTs on CPUs, FPGAs, and GPUs.

Asadi et al. introduce different realization schemes of tree-based models in the context of learning-to-rank tasks [26]. They introduce two different realization schemes, which will be discussed in more detail later: the first one uses a while-loop to iterate over individual nodes of the tree, whereas the second approach decomposes each tree into its if-else structure. For the first realization, the authors also consider a continuous data layout (i.e., an array of *structs*) to increase data locality but do not directly optimize each realization. Also note that the authors mainly consider gradient-boosted trees. There, the individual trees are usually "weak" in a sense, that they are comparably small, as opposed to larger trees in RFs.

Also in the context of ranking models, Lucchese et al. present the QuickScorer algorithm for gradient boosted trees [162, 450]. In this approach, the authors discard the tree structure and decompose each tree into its comparisons. Then, they sort the comparisons of the entire ensemble according to the feature value and perform them one after another instead of traversing trees in a classical sense. To do so, they introduce a 2^{Δ}-dimensional bit vector, where Δ is the height of a tree in which the most significant bit (MSB) signifies the prediction leaf node of that tree. This way, the algorithm can reuse comparisons across all ensemble members while minimizing cache misses. In [452] the authors further enhance their method by adding vectorization over multiple examples for more efficient batch-processing. To mitigate the limitations of a fixed height, Ye et al. propose in [721] using an encoding scheme called epitome that decodes the bitvectors on the fly while preserving vectorization. We note that, while these methods usually offer a tremendous speed-up, they execute *all* possible comparisons in the entire ensemble in the worst case. Thus, they are especially effective for large ensembles of smaller trees commonly produced by gradient boosting algorithms.

Kim et al. present in [373] a realization for binary search trees using vectorization units on Intel CPUs and compare their realization against a GPU realization. The authors provide insight on how to tailor the realization to Intel CPUs by taking into account register sizes, cache sizes, and page sizes. Their work is specialized for Intel CPUs, and thus, it is not directly applicable for different CPU architectures. Lucchese and colleagues have already noticed, that many nodes are seldom visited [450]. Buschjäger and Morik formalize this observation in [110] by estimating the probabilities of specific paths during tree traversal. Based on this probabilistic view of model execution or inference, the authors consider different realization schemes for tree traversal and theoretically analyze their runtime. Note, however, that this model of computation remains at the software level and does not include the memory layout. Buschjäger et al. enhance this model in [108] by including the memory layout in their model. They show how to minimize cache misses and how different realizations affect the instruction and data cache differently for executing ensembles of large trees commonly found in RFs. We will now discuss this paper in more detail.

7.3.3 A Probabilistic View of DT Execution

We consider supervised learning problems, in which we infer a model $f : \mathbb{R}^d \to \mathcal{Y}$ from labeled training data $\{(\mathbf{x}_i, y_i) | i = 1, \dots, N\}$ to predict the value $f(\mathbf{x})$ of new, unseen observations. For $\mathcal{Y} = \mathbb{R}$, we have a regression problem, for $\mathcal{Y} = \{0, 1, \dots\}$, we have a classification problem.

Tree ensembles train a set of individual trees and combine their predictions to establish a joint model. In the classical Random Forest (RF) approach by Breiman [72], K DTs are trained using different samples of input features. Other RFs variations have been explored, such as those that train trees on samples of data (bagging) [71] or those that randomly generate splits for training [250]. Boosting [610] also frequently uses decision trees as their weak base learners, but trains them sequentially to correct each other.

A decision tree is a simple, tree-structured model that consists of inner nodes with two children and leaf nodes. Each inner node compares the feature value x_f of the current sample $\mathbf{x}$ against a threshold t where f and t are computed during tree training. Depending on the outcome of this comparison, either the left or the right child of this node is used until a leaf node is found. The leaf node stores a constant prediction value (e.g. the estimated class probabilities that fall into the leaf) which is then returned.

Our goal is to analyze the probability of performing a certain comparison while traversing a DT. Based on this analysis, we can decide for each tree, which realization and which data layout is best. Our notation is the following: each node receives a unique identifier (e.g., in breath-first order) i. We denote the left child of i with $l(i)$ and the right child with $r(i)$. Note that every observation takes exactly one path $\pi(\mathbf{x})$ from the root node to one leaf. To lighten the notation, we drop the argument $\mathbf{x}$, if we are not

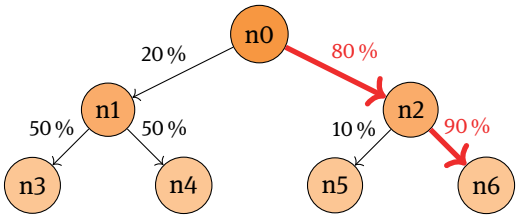


Fig. 7.12: Decision tree with probabilities of the path.

interested in the path of a specific observation. As established in [110], we model each comparison at node i as a Bernoulli experiment in which we take the path towards the left child with probability $p(i \to l(i))$ and towards the right child with $p(i \to r(i))$. It holds that $p(i \to l(i)) = 1 - p(i \to r(i))$. An example can be found in Figure 7.12.

The probabilities $p(i \to l(i))$ and $p(i \to r(i))$ can be estimated with the training data by counting the number of samples at each node i taking the left and right path. Assume a path of length L with $\pi = (i_1, i_2, \ldots, i_L)$, where i_{j+1} is either the left or the right child of the j^{th} node on the path. Following this path consists of a series of Bernoulli experiments, each with probability $p(i_j \to i_{j+1})$. Let $\mathcal{P}$ denote the set of all paths in the tree. The probability of taking path $\pi \in \mathcal{P}$ is given by

$$p(\pi) = p(i_0 \to i_1) \cdot \ldots \cdot p(i_{L-1} \to i_L) = \prod_{j=0}^{L} p(i_j \to i_{j+1}) \tag{7.13}$$

Again, let i be a node, there is exactly one path $\pi = (0, \ldots, i)$ ending in node i. We call the probability of the path leading to node i the probability of that node, that is $p(i) = p((0, \ldots, i))$. Let $\mathcal{T}$ be the set of all nodes in the tree. We define the probability for every subset of nodes $T \subseteq \mathcal{T}$ as:

$$p(T) = \sum_{i \in T} p(i) \tag{7.14}$$

7.3.4 Memory Locality and Tree Realization

As mentioned, tree ensembles can consist of millions of nodes that must be stored and managed in the main memory. Hence, the memory layout of tree ensembles is one of the most crucial aspects of efficient tree traversal. In order to mitigate the performance gap between the main memory and the processor, smaller and faster memory subsystems are often introduced in modern computer architectures to hide the long read/write latency, in the forms of cache and scratchpad memories. Here we focus on the cache memory, which is commonly equipped in modern computing systems.

The cache memory basically acts as a buffer between the main memory and the CPU and stores the data and instructions that the CPU uses more frequently. This way,

frequently accessed parts of the memory can be loaded from the smaller, but much faster cache memory to reduce the latency of memory accesses. However, any misuse of cache memory might be even worse than no cache in the memory hierarchy because one cache miss triggers two loading instructions, one from the main memory to the cache and one from the cache to the processor. There are three types of cache misses [183]:

Compulsory misses are due to the first access to a memory block that the cache did not yet have a chance to buffer.

Capacity misses occur when some memory blocks are discarded from the cache memory due to the limited capacity, i.e., the program is working on more data than the cache capacity.

Conflict misses occur in set-associative or direct-mapped caches when several blocks are mapped to the same cache set.

The basic assumption of a cache is that of *memory localities*:

Temporal locality Recent data will be accessed in the near future, say, in small program loops.

Spatial locality Data at addresses close to the addresses of recently accessed data will be accessed in the near future, say, in sequential accesses to elements of an array.

These are the general assumptions for cache design, but please note that knowing how the caches exactly behave is difficult or even impossible. Caches are manufactured as parts of the closed IP of CPU manufacturers and hence the exact design of caches is unknown. Additionally, due to the fact that there are often competing processes running on a single CPU it is difficult to predict the cache behavior deterministically. In this contribution we suppose that the design of cache behaviors cannot be changed. The question we address is this: **How to realize a cache-friendly execution while preserving the functional behaviors of a given DT?**

First, we analyze the memory usage of two common realizations of DT, i.e., native Tree and If-else Tree that do not exploit the memory locality during the execution over the structure of DT. Then we discuss how we can make these two realizations more cache-friendly.

Native Tree The native tree implementation uses a while-loop to iterate over the individual tree nodes that are stored within a continuous data structure, say, in a one-dimensional array. An example code can be found in Listing 7.1. Although the usage of the simple loop with a few lines of codes preserves the temporal locality, the accesses over the nodes of a DT do not have spatial locality. The nodes are often allocated sequentially according to the indexes, whereas such indexes might not take the execution of the DT into consideration, e.g., the nodes on one path might not be allocated sequentially. In addition, if the distance between each node of the path is greater than the number of nodes that can be hosted into a cache set, some nodes will

be loaded into caches but not used at all, leading to much *capacity and conflict cache misses*.

Listing 7.1: Example for native tree structure in C++.

```
struct Node {
  bool isLeaf;
  unsigned int prediction; // Predicted label
  unsigned char feature; // Targeted feature
  float split; // Threshold
  unsigned short leftChild, rightChild;
};
Node tree[] = {{0,0,0,8191,1,2},{0,0,1,2048,3,4},..]}
bool predict(short const x[3]){
  unsigned int i = 0;
  While(!tree[i].isLeaf) {
    if (x[tree[i].feature] <= tree[i].split) {
      i = tree[i].leftChild;
    } else {
      i = tree[i].rightChild;
    }
  }
  return tree[i].prediction;
}
```

If-Else Tree An alternative is the if-else tree, which statically encodes the split values of nodes in the instructions. This realization essentially avoids the indirect memory accesses required by the native tree and usually improves the runtime efficiency significantly. An example code can be found in Listing 7.2. However, the advantage of the temporal locality in the instruction cache might be completely abandoned. Since DTs are naturally composed of many branches, some encoded instructions might be prefetched into the instruction cache but not used. Additionally, if the size of the instructions for one DT is greater than the size of the instruction cache, the cached instructions may be evicted out by loading other instructions due to the *capacity and conflict cache misses*.

Listing 7.2: Example for if-else trees in C++.

```
bool predict(short const x[3]){
  if(x[0] <= 8191){
    if(x[1] <= 2048){
      return true;
    } else {
      return false;
    }
  } else {
    if(x[2] <= 512){
      return true;
    } else {
      return false;
    }
  }
}
```

7.3.5 Memory Layout Optimization

In the following, we analyze the caching behaviors of the two different realizations and present our tree-framing algorithms to optimize the memory layout at the application layer accordingly.

Native Tree As shown in Listing 7.1, a DT can be realized by allocating the tree nodes sequentially in an 1-D array, and a simple loop can access them according to the comparison between the feature and the split value. We first observe that, in fact, half of the nodes in a tree are leaf nodes storing a prediction value. This naive realization, however, assumes the same data type for each node, incurring unnecessary memory usage. Second, the access pattern of a DT forms a unique path from the root to a leaf for each input data, but the nodes are typically sequentially allocated in the array according to Breadth-First Search (BFS).[5] The distance between each accessed node becomes larger when the accessed nodes are placed deeper in the DT. The proposed optimization is twofold: 1) reducing compulsory cache misses by encoding the predicted label into the field of children, and 2) reducing capacity and conflict cache misses by allocating as many nodes as possible from the same path into the same cache set.

When a node is loaded, the following nodes in the array are prefetched into the data cache sequentially. If the size of memory for each node can be reduced, more nodes can be loaded into the cache at once so that overall compulsory cache misses can be reduced. To reduce memory consumption we can completely remove the `isLeaf`

5 Please note that the problem is not limited to BFS. Here we point out the demand of considering the access pattern when allocating nodes to memory.

and `prediction` fields, and store the predicted labels of the children directly in the respective fields by encoding the node type with an indicator field, i.e., removing one Boolean variable and two unsigned shorts by adding one unsigned short.

As mentioned earlier, the sequence of stored nodes is not consistent with the access pattern over the execution of the tree, so the benefit of caching cannot be utilized properly. A sensible solution is to leverage the probabilistic view on DT execution to identify nodes that were likely executed consecutively and place them in memory accordingly. Let τ be the cache set size and A be the array in which we place all nodes of $\mathcal{T}$. Furthermore, let $\mathcal{C}$ be the candidate list of nodes in $\mathcal{T}$ that have not been placed in A yet and let $\mathcal{S}$ denote the nodes that should be placed in the same cache set. For each node, we greedily choose a child that has the highest probability on the current path and place it in $\mathcal{S}$. Once $\mathcal{S}$ contains $\tau - 1$ elements (and hence is full), we append all nodes from $\mathcal{S}$ to the array A, clean up $\mathcal{S}$, and repeat the above procedure for the next set. The details are summarized in Algorithm 6.

Algorithm 6: Optimized path layout
Data: Tree-nodes $\mathcal{T}$, maximum nodes per set τ
Result: A data array A with the path-oriented layout

```
1  A = [ ]
2  C ← {0}
3  while C ≠ ∅ do
4      i ← arg max_{j∈C}{p(π(j))}
5      C ← C \ {i}
6      S ← {i}
7      while |S| ≠ τ do
8          if i is leaf-node and C ≠ ∅ then
9              i ← arg max_{j∈C}{p(π(j))}
10             C ← C \ {i}
11         else
12             C ← C ∪ arg min{p(i → l(i)), p(i → r(i))}
13             i ← arg max{p(i → l(i)), p(i → r(i))}
14             if |S| = τ − 1 then
15                 C ← C ∪ {l(i), r(i)}
16         S ← S ∪ {i}
17     A.append(S)
```

Please note that adding a new node to $\mathcal{S}$ (Line 7) has two possible actions for the encoding procedure:

- The current node is a split node. The algorithm picks the next node based on the children's probabilities and puts a more probable child in $\mathcal{S}$ and the other children into the candidate list $\mathcal{C}$.
- The checked node is a leaf node, i.e., the end of the path. The algorithm picks a sub-root with the highest probability from the candidate list $\mathcal{C}$ as long as it is not empty. The traversal starts again until $\mathcal{S}$ is full.

If the current $\mathcal{S}$ is full, but a path is not finished yet (Line 14), two children of the current node are returned to the candidate list $\mathcal{C}$ (Line 16). A sub-root that has the highest probability is picked from $\mathcal{C}$ for the next new set $\mathcal{S}$. The algorithm outputs the optimized memory layout over nodes in which path-oriented sets are sequentially allocated to the array.

If-Else Tree As shown in Listing 7.2, a DT can be realized by unrolling the comparisons of a DT into conditional statements with the if-else blocks. This version avoids the indirect memory accesses and does not consider the execution pattern of a DT. The proposed optimization is also twofold: 1) reducing compulsory cache misses by reducing the branch executions, and 2) reducing capacity and conflict cache misses by grouping those nodes used most of the time, e.g., the root node.

When a compulsory cache miss takes place, several consecutive instructions are fetched into the instruction cache, even though some of them might not be executed due to branches. An analysis of the corresponding assembly code reveals that only the branches for else statements are generated in general. In order to increase the chance of using prefetched instructions, the possibility of branch executions should be reduced. Towards this, we propose traversing all paths in the DT and swapping the children of every node i when $p(i \to l(i)) \geq p(i \to r(i))$.

Furthermore, unlike the native tree, the positions of unrolled nodes cannot be freely allocated. The size of nodes from a DT is likely greater than the size of the instruction cache. Because of the capacity and conflict cache misses the cached instructions may be evicted by fetching other instructions. We propose partitioning nodes into different computation kernel functions, and leveraging `goto` statements to break the tie between if-else blocks so that we can put probable nodes together.

Let $\mathcal{K}$ denote the kernel function and let $s(i)$ be a mapping function returning the instruction size of node i. We formulate the following optimization problem:

$$\mathcal{K} = \arg\max \left\{ p(T) \middle| T \subseteq \mathcal{T} \text{s.t.} \sum_{i \in T} s(i) \leq \beta \right\}, \tag{7.15}$$

where β is a given budget related to the size of the instruction cache on the targeted architecture. Given $\mathcal{K}$, these nodes likely remain in the cache once they are fetched, whereas the remaining nodes $\mathcal{L} = \mathcal{P} \setminus \mathcal{K}$ may be evicted more often. In order to avoid iterating over all possible subsets of $\mathcal{T}$, which might be computationally inefficient, we propose a greedy algorithm to partition nodes in a path-wise manner, summarized in

Algorithm 7. At first, the algorithm swaps the children according to their probabilities,

Algorithm 7: Optimized *if-else* tree
Data: Tree $\mathcal{T}$, Paths $\mathcal{P} = \{\pi_1, \ldots, \pi_M\}$
Result: Kernel $\mathcal{K}$, Label $\mathcal{L}$

```
swapChildren(T)
P ← sortByProbabilities(P)
b ← 0
for π ∈ P do
    for i ∈ π do
        if b + s(i) > B then
            Add i to L
        else
            Add i to K
            b ← b + s(i)
```

and sorts all paths in the tree by their probabilities. Afterwards, the approach greedily appends nodes one by one into $\mathcal{K}$ until the accumulated size of the added nodes b is greater than the given budget $\mathcal{B}$. The rest of the nodes are all added to $\mathcal{L}$. Once the nodes are grouped into $\mathcal{K}$ and $\mathcal{L}$, we can use `goto` statements to break the sequential generation of if-else blocks. First, we generate if-else blocks for all nodes in $\mathcal{K}$. Once the left/right child of one of those nodes is in $\mathcal{L}$, a `goto` statement is generated at the same position to replace the original if-else statement. Then, the corresponding if-else statements of this node and its children are all generated into a label block at the end, which is branched from the goto statement. Listing 7.3 shows an example based on Listing 7.2 by applying Algorithm 7.

Listing 7.3: If-else structure in C++ with goto statements.

```
bool predict(short const x[3]){
  if(x[0] > 8191){
    if(x[2] <= 512){
      return true;
    } else {
      return false;
    }
  } else {
    goto Label0;
  }
Label0:
  {
    if(x[1] <= 2048){
      return true;
    } else {
      return false;
    }
  }
}
```

The remaining question is how to estimate the instruction size $s(\cdot)$ of each node. In general, the instruction set size differs for two different types of nodes:

Split nodes require three types of instructions. First, the values of the target feature and the corresponding threshold are loaded into registers. Second, the values inside the registers are compared against constant values. Last, a jump out of the current block is performed based on the comparison.

Leaf nodes need two types of instructions. First, the return value of the prediction is stored in a register, and second, a jump back to the caller of the if-else tree is performed.

Therefore, we can estimate $s(\cdot)$ by counting the number of generated instructions for a tree node. Table 7.7 summarizes the expected size of instructions for ARM, X86 (Intel), and PPC in an isolated example.[6] Please note that in a real application, the actual number of instructions depends on the adopted compilation tool-chains and the actual realization. An advanced automation can be further explored by exploiting compiler features, e.g., annotations on the source code, to enforce the executing patterns. By doing so, the number of generated instructions can be firmed in the proposal algorithm as for example done in ongoing research such as [132].

6 We adopted GNU C++ (g++) compiler version 4.8.3 for ARM, version 4.9.2 for PPC, and version 5.4.0 for Intel with `-O0` option.

Tab. 7.7: The expected size of instructions for a split node and a leaf node in a decision tree on ARM (`Raspberry PI 2`), PPC (`NXP T4240 processors`) and Intel (`Intel Core i7-6700`) processors.

	ARM [Bytes]		PPC [Bytes]		Intel [Bytes]	
Type	Int	Float	Int	Float	Int	Float
Split	20	32	20	48	28	17
Leaf	8	8	8	8	10	10

7.3.6 Architecture-Aware Code Generator

As noted earlier, for each combination of tree ensembles and target hardware architecture a different implementation might offer the best inferencing solution. Hence, we implement the discussed tree-framing methods in a single code-generator framework that generates the optimized realizations for a given forest and target platform. Figure 7.13 gives an overview of the whole workflow. First, the pre-trained forest (in a `JSON` format) is loaded. Afterwards, the corresponding intermediate representation of the ML model is generated, and the proposed optimizations are performed, e.g., branch swapping, node re-indexing, etc. Finally, we provide a set of C-style templates that represent the specific implementation types (e.g. *native* or *if-else*). Several auxiliary scripts scripts are provided to automate the above procedures, e.g., selecting corresponding cross-compilers. Per default sci-kit learn models are targets [561] but other model definitions, in, say, the ONNX format are also supported. More details can be found at https://github.com/sbuschjaeger/fastinference.

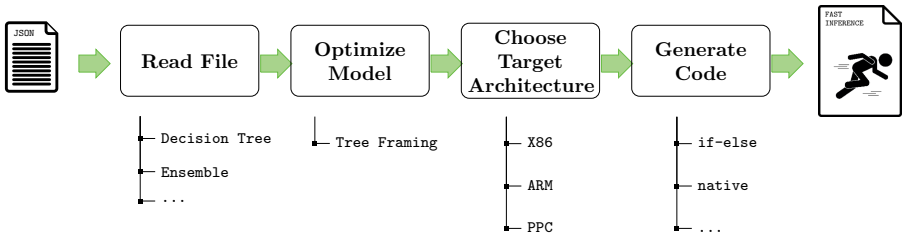


Fig. 7.13: Workflow of our code generator. The model configuration is loaded into an internal representation. If selected, optimizations are performed on the model before code generation. Afterwards, the target architecture and the appropriate templates are selected for final code generation.

7.3.7 Experimental Evaluation

We have performed 1800 different experiments by training Decision Trees (DT) [73], Random Forests (RF), [72] and Extremely randomized Trees (ET) [250] on 12 different datasets with varying tree-depths to generate the aforementioned realizations for different architectures, i.e., X86, PPC, and ARM CPUs. Table 7.8 shows the datasets we used during the experiments. All datasets are available in the UCI Machine Learning Repository [31] except for MNIST [420], IMDB [456], and FACT [17]. In addition to the number of features and the number of examples during test time, we also report the range of accuracy for the three different models DT, RF, and ET. In all experiments we used the CART algorithm with the Gini score criterion for node-splitting and trained models using the `sklearn` package[561]. For RF and ET, we used 25 trees. If the respective dataset comes with a pre-computed train/test split, we use this. Otherwise, we use 75 % of the data for training and 25 % of the data for testing. DTs often do not achieve high accuracy, whereas RF and ET perform best with large trees. We did not perform any hyperparameter optimization with respect to the classification accuracy and report the accuracy here to validate our code generator.

Since `sklearn` is arguably one of the most-used machine learning libraries we also compared its performance against our implementation. We found that, our realization is on average 500 – 1500 times faster than `sklearn`. However, we admit that this comparison is biased, because large parts of `sklearn` are written in Python and optimized for batch execution. Thus, we excluded these comparisons in the following discussion. For space reasons, we focus our evaluation on RF models, but found that DT and ET result in similar behaviors across all systems. We use the *naive native* realization as the baseline for all experiments, and measure the average speed-up for each dataset of each optimization against this realization. To minimize unfairness due to caching, we classify all samples in the test set twice, but only report the runtime of the second run. We repeat the whole process 50 times and report the average speed-up across these 50 repetitions.

For *native* optimizations, we choose $\tau = 25$ on X86, $\tau = 8$ on ARM, and $\tau = 8$ for the PPC CPU. For *if-else* optimizations, we use an instruction-cache size $\beta = 128\,000$ bytes on X86, $\beta = 32\,000$ bytes on ARM, and $\beta = 32\,000$ bytes on the PPC CPU. The experiments were performed on an `Intel Core i7-6700` desktop machine with 16 GB RAM for `X86`. For `PPC`, we use a `NXP Reference Design Board` with `T4240 processors` and 6 GB RAM. For `ARM`, we use an `Raspberry PI 2` with an `ARMv7` CPU and 1 GB RAM.

Experiments on the X86 CPU Architecture Figure 7.14 depicts the average speed-up of the four different optimizations on Intel. First we note, that the *if-else* trees are the fastest on Intel and offer a speed-up of around three across all tree depths. For smaller tree depths from 1 – 10, we see that optimizing *if-else* trees only offers marginal speed-up. However, for larger tree depths of around 15 and 20, we can see that optimized

Tab. 7.8: Summary of datasets for our experiments based on UCI datasets [31], IMDB [456], MNIST [420], FACT [17].

Dataset	# Examples	# Features	Accuracy
adult	8141	64	0.76 - 0.86
bank	10 297	59	0.86 - 0.90
covertype	145 253	54	0.51 - 0.88
fact	369 450	16	0.81 - 0.87
imdb	25 000	10 000	0.54 - 0.80
letter	5000	16	0.06 - 0.95
magic	4755	10	0.64 - 0.87
mnist	10 000	784	0.17 - 0.96
satlog	2000	36	0.40 - 0.90
sensorless	14 628	48	0.10 - 0.99
wearable	41 409	17	0.57 - 0.99
wine-quality	1625	11	0.49 - 0.68

if-else trees can retain their speed-up and outperform unoptimized *if-else* trees with a speed-up factor larger than 3.

Native trees do not perform as well as *if-else* trees on Intel CPUs. Overall, the speed-up compared with *naive native* trees is only marginal for smaller trees below depth 15. Here, both versions, i.e., the *StandardNativeTree* and the *OptimizedNativeTree*, offer a speed-up of 1.5 at most. Interestingly, for larger trees around depth 15 and more, we again notice that our optimizations improve performance.

Experiments on the PPC CPU architecture Figure 7.15 depicts the average speed-up of the four different optimizations on PPC. We can observe that the results here are similar to Figure 7.14, in which *if-else* trees always outperform *native* trees with a speed-up in the range from 2 – 5. Along with the increment of tree depth, the speed-up from both *if-else* tree versions drops. Un-optimized *if-else* trees suffer especially from degraded performance, dropping to almost 2, whereas the optimized version can retain a speed-up of around 3.5.

Similar to X86 CPU, the *native* realization does not seem to be the best choice as it provides a speed-up under 2 in all cases. However, with increasing tree depths, optimizations are more important. It is worth noting, that we can observe cases where the *native* trees outperform *if-else* trees when tree depth is larger than 15.

Experiments on the ARM CPU Architecture Figure 7.16 depicts the average speed-up of the four different optimizations on ARM. We observe that the situation on ARM is more fragmented than that of X86 and PPC. In general, we are able to achieve a speed-up of around 4 for small trees, which drops to around 2 – 3 for larger trees. Both realizations roughly start with the same speed-up factor for small trees, but then quickly diverge for tree depth from around 5 – 15. In this range of tree depth, we see that *if-else* trees are the

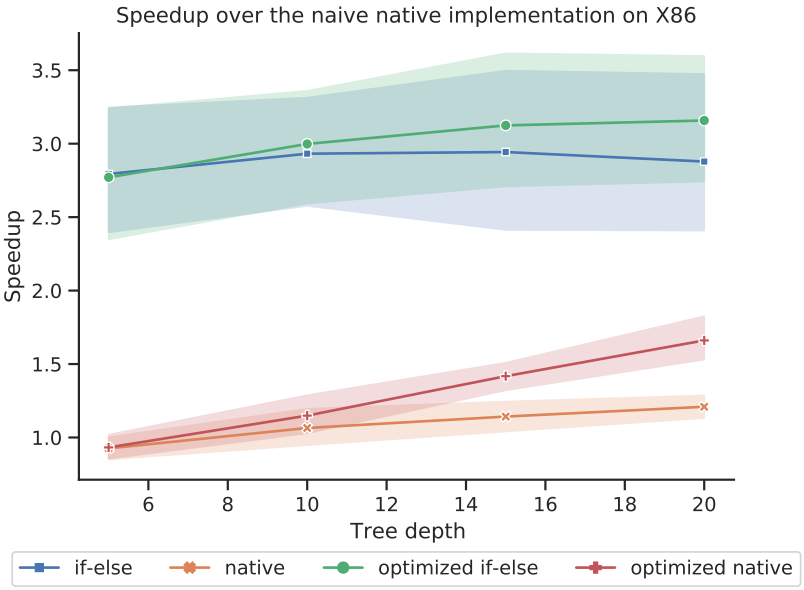


Fig. 7.14: Average speed-up factor for real-time execution compared with the naive native realization on Intel for tree depths from 1 – 20.

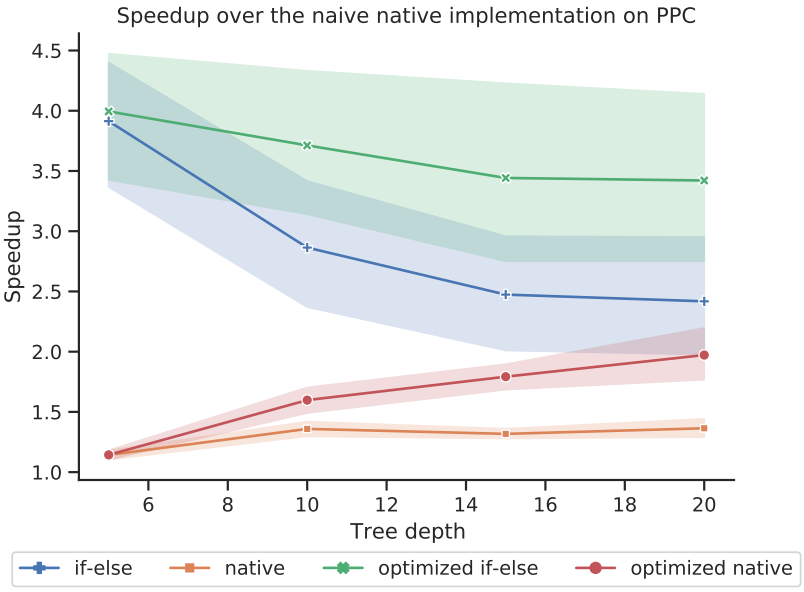


Fig. 7.15: Average speed-up factor for real-time execution compared to the naive native realization on PPC for tree depths from 1 – 20.

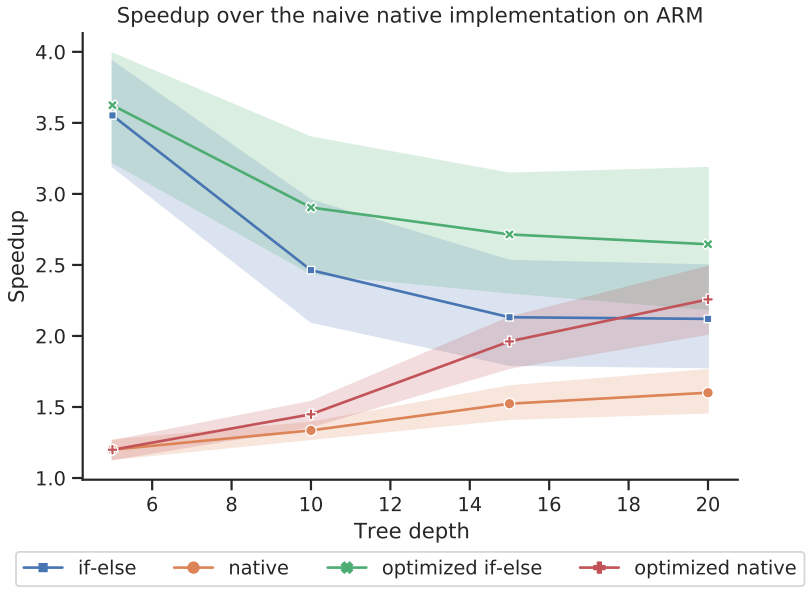


Fig. 7.16: Average speed-up factor for real-time execution compared with the naive native realization on ARM for tree depths from 1 – 20.

fastest choice on ARM. Additionally, we notice that with increasing tree depths cache optimizations become more important and consistently outperform their un-optimized counterpart. Once trees are sufficiently large, we see that the *native* trees match again the performance of *if-else* trees and even outperform them for tree depths of 15 and 20 in some cases. In this sense, the results are similar to what we have seen on the PPC architecture.

7.3.8 Discussion of the Experiments

The experiments show differences and similarities across the three architectures. Here, we want to discuss these phenomena in terms of the properties of the specific architectures, as well as the particular CPU models used for experiments. We note that one of the main architectural differences between X86, ARM, and PPC are the available instructions. Since *native* trees only use a small amount of hot-code, the differences between CPU architectures will likely not matter much here. However, while looking at *if-else* trees, we can expect a larger difference. To further investigate the interplay between

CPU architectures and code size, we consider Table 7.9, [7] which depicts the instruction size of a tree kernel function for varying tree depths over the FACT dataset (containing floating-point features) and the covertype dataset (containing integer features) under the standard *if-else* tree realization. For Intel CPUs, as shown in Figure 7.14, we notice

Tab. 7.9: The actual size of instructions for *if-else* tree executing kernel functions on different architectures with the O3 option.

(a) Kernel size with integer features for covertype dataset

DateType	DT-1	DT-5	DT-10	DT-15	DT-20
Intel	224	575	8185	51005	167644
PPC	232	604	7732	51840	170772
ARM	204	604	9040	55012	180628

(b) Kernel size with floating point features for fact dataset

DateType	DT-1	DT-5	DT-10	DT-15	DT-20
Intel	96	415	17023	127330	404722
PPC	96	556	20996	169696	577952
ARM	88	428	18436	154992	542020

that *if-else* trees are the best choice. There are mainly two reasons. First, `X86` CPUs are Complex Instruction Set Computers (CISC) offering a very rich set of instructions that include all sorts of specialized operations. Since *if-else* trees unroll the complete tree structure into instructions, they give the compiler the opportunity to utilize this multitude of instructions to the fullest by encoding larger parts of the tree in single instructions. From Table 7.9 we can also see that the Intel CPU almost always requires the fewest instructions per decision tree. Second, in our experimental setting, the `Intel Core i7-6700` CPU has a comparably large instruction cache of 256 KiB combined with two larger shared caches of 1 MiB (L2 Cache) and 8 MiB (L3 Cache). Thus, by encoding a single tree in only a few instructions, it is likely to fit it into the larger instruction cache. By contrast, *native* trees do not benefit from the CISC architecture and require additional space in the data cache by encoding the tree nodes as data instead of instructions.

As with the `X86` architecture, we have seen that *if-else* trees perform very well on the `PPC` architecture, but to a lesser extent. The `PPC` CPU architecture is a Reduced Instruction Set Computer (RISC) with performance enhancement for high performance computing. RISC does not offer instructions for specialized operations as CISC does.

7 Although the instructions generated by the compiler may differ due to aggressive compiler optimization (O3) compared with the presented node sizes (O0 optimization) in Table 7.7, the code generator at the end selects the O3 option to accelerate the realizations as much as possible.

Thus, the compiler must largely rely on the combination of (comparably) simple instructions to implement *if-else* trees. This, in turn, results in larger code that is less likely to fit into the instruction cache. Comparing the instruction size of PPC with X86 in Table 7.9 we see that the PPC architecture indeed requires more instructions than with X86. Interestingly, this case is less severe for integer features, due to the enhancements in this instruction set architecture. Considering the cache sizes of the T4240 processors used in the experiments, we see that it only has a 32 KiB instruction cache, but also comes with a 2 MiB shared L2 cache, which is even larger than the Intel Core i7-6700 CPU. For smaller trees of around 5 – 10, the cache sizes are still large enough to hold all trees, and thus *if-else* trees are still the fastest choice. If trees become large (depth 10 or more), the instruction cache is not enough to hold all trees and we must rely on the larger L2 cache. However, this cache is slower, which in combination with the larger code size explains the performance drop for larger trees.

Finally, we discuss the fragmented behavior of the ARM architecture. Much like its PPC counterpart, ARM also uses a RISC architecture. However, ARM's RISC does not come with specialized instructions for high-performance computing, and thus the compiler has to completely rely on the combination of simple instructions for *if-else* realization. This in turn results in even larger code for integer features, which is less likely to fit into the instruction cache as shown in Table 7.9. Interestingly, for floating-point features, we see that the ARM CPU uses fewer instructions than the PPC CPU, which is attributable to the specific CPU model used during experiments. The T4240 processors are optimized for high-performance computing in a low-power embedded computing setting, such as networking applications, and thus are optimized for integer operations. By contrast, the ARMv7 CPU of the Raspberry PI 2 is a general-purpose CPU aimed at the needs of the average user, and thus it places a larger emphasis on floating-point operations compared with the T4240 processors. It has a 32 KiB instruction cache in combination with a significantly smaller 512 KiB L2 shared cache. Compared with the other CPUs, this means that the ARM CPU has 2 – 16 times less L2/L3 cache available. For smaller trees around a depth of 5 – 10, the cache sizes are still enough to hold all trees, and thus *if-else* trees are still the fastest choice. For larger tree depths, however, the instruction cache is not enough and *native* structures using the data cache become faster. However, since the data cache is also small, both caches are filled quickly to their maximum. Interestingly, if we optimize both *if-else* and *native* trees, we end up with roughly the same performance.

7.3.9 Conclusion

DTs form one of the building blocks of modern machine learning and ensembles of decision trees are one of the most successful classifiers regularly achieving state-of-the-art performance in real-world applications. DTs are generally regarded as 'simple'

classifiers that can be executed even on the tiniest of hardware. However, a tree easily contains up to millions of decision nodes that must be stored and managed which can be a challenge even for large server hardware. Cache memory is commonly adopted in today's von-Neumann computing architecture to hide the long latency between the main memory and the processor. Hence, an efficient realization of a given tree ensemble must respect this memory hierarchy and provide a suitable memory layout of the decision nodes for optimal performance. In every modern programming language there are at least two ways to implement a DT: either one decomposes the tree into its if-else structure or one uses a while-loop to iterate over a continuous array of nodes. Both approaches offer different caching behaviours that can be further enhanced by the tree-framing methods discussed in this contribution. At the core of these methods lies the fact that DTs do not have a deterministic runtime, but its execution time may vary depending on the current sample. Hence, a probabilistic view of DT execution estimates the most probable paths of the tree and frames the tree so that these paths are likely to remain in the cache. The experimental evaluation shows a speed-up around $2 - 4$ across three different hardware architectures on a variety of datasets without any loss in accuracy occurs.