Efficient Hierarchical Decomposition of Repetitive Traces for ML-Driven Analysis

Johannes Knödtel, Marc Reichenbach
Institute of Applied Microelectronics and Computer Engineering, University of Rostock
{johannes.knoedtel,marc.reichenbach}@uni-rostock.de

Abstract

In modern computing architectures, the execution of programs often results in highly repetitive and structured linear sequences of microarchitectural states and instruction traces. This is especially true for most signal and image processing task, such as stencil codes, convolution and most ANNs, due to the loop-oriented nature of their codes. When applying machine learning (ML) algorithms to such sequences, especially for tasks like predicting power consumption on specific CPUs, it becomes inefficient to process every repetitive sequence in its entirety. This inefficiency arises from the fact that, beyond a certain point, the architecture reaches a steady state, rendering further processing redundant. This makes finding a repetition representation of such data worthwhile in both the training and execution phases of these models. In this work we present a method to identify hierarchical structures, such as nested loops, and accurately detects n-fold repetitions beyond simple duplication. By leveraging rolling hashes, our algorithm achieves a sufficiently high performance in the relevant cases. For large problem sizes, the approach is highly parallelizable, resulting in a significant reduction in runtime. Depending on the input data, it is able to demonstrate excellent scalability, achieving a speedup of over 152 compared to the serial version on a server with 192 physical cores, in one of the examples presented in this paper. This advancement paves the way for more efficient ML-based analysis of program behavior on complex architectures. We offer two open source implementations of this method: A more easily comprehesible Python variant and a highly optimized and parallel C version.

1 Introduction

With the rise of ML, applying it to estimate properties of the execution of software on specific hardware platforms has become inevitable, for instance in [1] for energy estimation and in [2] for runtime prediction. Typically, executed instructions or simulated microarchitectural states and eventssuch as cache misses, hazards, memory latency, and othersare recorded for further analysis. Cyclic behavior often becomes immediately apparent in this data. This observation aligns with a well-known empirical rule frequently mentioned in computer science lectures on compilers: 90% (occasionally cited as 80%) of a program's execution time is spent repeatedly executing loop bodiesa phenomenon noted by Donald Knuth as far back as 1974 [3].

For example, we ran a stencil code on a RISC-V emulator and traced the types of FPU instructions executed. As shown in Fig. 1(a), this trace reveals a highly structured sequence, characteristic of the loop-based structure of the program. Such structured sequences often contain significant redundancy, which can pose challenges in machine learning applications. Excessive redundancy risks overfitting and leads to computational inefficiency during both training and inference [4].

It is evident that unrolled sequences, such as the one in Fig. 1(a) can be translated into a hierarchical representation, as illustrated in Fig. 1(b). This transformation significantly reduces redundancy. For instance, in analyzing the first 10^6 instructions of our sequence, the hierarchical representation contains fewer than 500 unique instructions, not counting the repetitions themselvesamounting to approximately 0.05% of the original sequence.

In the context of machine learning, such hierarchical representations can be weighted based on the degree of repetition, enabling efficient training that proportionally emphasizes essential sections of the data while minimizing redundancy. To support this, we present an algorithm capable of efficiently detecting nested repetitions and generating a repetition-aware representation of the input data. The main requirement to adapt such a method is that the input data needs to be mapped to a sequence of symbolic elements (e.g. "`add r1, r2; sub r2, r3; sub r2, r3`" $\rightarrow$ "`abb`"). In most scenarios this should be a one to one mapping, so that it can be reversed after transforming it to a hierarchical representation.

We provide two open-source implementations: a Python version optimized for clarity and a C version optimized for performance. Both implementations offer APIs to facilitate integration into the aforementioned use cases. The C implementation includes a native C API as well as Python bindings, while the Python implementation provides a native Python interface only.

The algorithm primarily relies on rolling hashes to efficiently identify repeated sections, progressing from larger to smaller subsequences. For the detected repetitions, the algorithm is applied recursively, reducing the theoretical best-case complexity to $O(N)$. For large problem sizes, the algorithm exhibits a high degree of parallelism, which can be leveraged in a straightforward manner with substantial parallel efficiency. On a server with 192 physical cores, we

Authorized licensed use limited to: Centro Universitário Fei. Downloaded on June 23,2025 at 20:18:27 UTC from IEEE Xplore. Restrictions apply.

$csc\,s\,c\,s\,c\,s\,c\,s\,c\,s\,c\,s\,c\,s\,c\,s\,c\,s\,c\ldots l$
$lllllllllllll\,aaaaaaa\,lllllll\,aaaaaaa\,ll\,aaaaaaa\,mmmmmmmm\,ll\,sssssss\,lllllllllllllllll\ldots$
$lllllllllllll\,aaaaaaa\,lllllll\,aaaaaaa\,ll\,aaaaaaa\,mmmmmmmm\,ll\,sssssss\,lllllllllllllllll\ldots$
$\ldots$

l	Load	a	Add
s	Store	m	Multiply
c	Conversion		

(a) Original Instruction Sequence

$(cs)^{1024}l$
$(l^7(l^7a^7)^2l^2a^7m^7l^2s^7l^{19}a^{12}l^2a^8m^8(ls^2)^2s^4l^{15}a^{12}l^2a^8m^8(ls^3)^2s^2l^{12}la^7l^2(l^2a^7)^2m^7s^7)^{256}$
$\ldots$

(b) Hierarchical Representation

Figure 1 Example of a Hierarchical Decomposition of a Instruction sequence

achieved a speedup of over 152 for the use case illustrated in Fig. 1. However, this behavior depends on the input data, as discussed in Section 4.

The remainder of this paper is structured as follows: Section 2 provides an overview of related algorithms and methods, with a comparison to our proposed approach. Next, we present the details of the proposed algorithm in Section 3, followed by an analysis of its best- and worst-case runtime complexities in Section 4. This section also includes performance measurements on two synthetic datasets to evaluate the algorithms practical efficiency. Finally, in Section 5, we summarize our contributions and suggest potential optimizations as directions for future work.

2 Related Work

We have identified two specific fields that address similar challenges: the detection of tandem repeats and the compression of traces to reduce file size. The problem of identifying tandem repeats arises in various domains, most notably in bioinformatics. Meanwhile, efforts to address redundancies, albeit not explicitly targeting repetitions, are prevalent in trace compression algorithms. The following subsections provide a summary of the state-of-the-art in these respective fields.

2.1 Tandem Repeats

A specific and extensively problem is the detection of tandem repeats, defined as contiguous sequences of identical substrings, such as `abc` in `abcabcabc`. In bioinformatics, tandem repeats are a key feature often analyzed in DNA sequences. This problem can be efficiently solved in linear time using suffix trees [5], whereas earlier approaches typically achieved a complexity of $O(N \log N)$ [6, 7].

However, tandem repeats differ from the problem of representing nested repetitions. To our knowledge, the specific challenge of hierarchical repetition detection has only been addressed by Nakamura et al. [8]. The authors propose several algorithms, two of which are particularly relevant to our work: CMRT, with a time complexity of $O(N^3)$, and CMRT-C, with a complexity of $O(n^2 \log N)$. These algorithms are notably intricate, relying on suffix tree operations described in earlier work.

Although an implementation by the authors appears to be available, it is hosted in a GitHub repository that is neither referenced in the original publication nor officially released until years after its publication. The repository lacks documentation, making it unclear which of the proposed algorithms was implemented, and its absence of a license renders it unsuitable for our intended applications.

Due to the overlap with string processing, we will use terminology from string algorithms, such as the ones presented in this subsection, in the following sections to align with established methods and related concepts.

2.2 Trace Compression

Another closely related field is trace compression, which focuses on minimizing the storage requirements of large instruction traces. This objective differs from our goal of hierarchically decomposing traces across various microarchitectural aspects, but still holds relevance to our work.

For instance, Burtscher et al. introduced the VCP format, which leverages value prediction techniques to compress individual trace fields [9, 10, 11]. Their approach predicts values by identifying patterns in previous entries, such as strided or incrementing sequences, or by reusing the most recently observed value. While these methods are particularly effective at detecting operand patterns in executed instructions, they are highly specialized for this purpose and not directly tailored for machine learning applications. Nonetheless, with appropriate modifications, these techniques could be adapted to complement our approach, either as a preprocessing step or as a postprocessing enhancement to hierarchical repetition detection.

3 Proposed Algorithm

In this section, we present our proposed algorithm following a bottom-up approach.

Authorized licensed use limited to: Centro Universitário Fei. Downloaded on June 23,2025 at 20:18:27 UTC from IEEE Xplore. Restrictions apply.

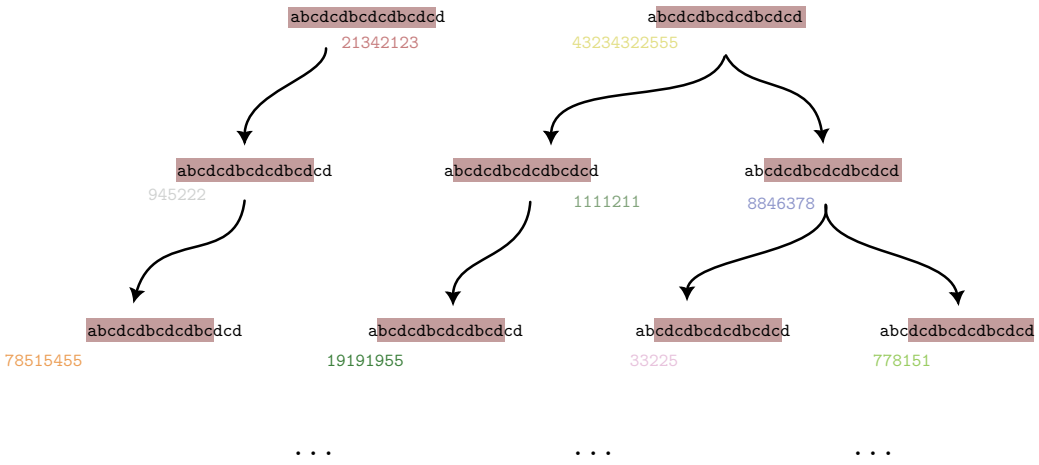


Figure 2 Calculation of hashes. Values are only for illustration purposes.

3.1 Rolling Hashes

A key aspect of our algorithm's high performance lies in its use of rolling hashes. Rolling hashes operate on sliding windows within a string, enabling the computation of a new hash for each shifted window in constant time. Our implementation utilizes a polynomial function as defined in the Rabin-Karp string search algorithm [12], though other rolling hash techniques could also be adapted for this purpose. Given a arbitrary fixed base b and large prime M the hash h for a string x in the bounds n and m is defined as:

$$h(x,n,m) = \sum_{i=n}^{m} x_i b^{i-n} \mod M$$

In a traditional rolling hash, the hash window slides over the input string one position at a time. However, our algorithm requires the ability to contract the window, reducing its size from either the left or the right. To contract the window from the right, we subtract the polynomial's constant term and multiply by the multiplicative inverse. This approach accounts for the fact that division does not distribute over the modulo operation. Multiplying by the multiplicative inverse effectively reduces the polynomial's order by one. This value can be precomputed efficiently for the chosen b and M.

Contracting the window from the left is straightforward, as we simply subtract the highest-order term:

$$h(x,n,m-1) = (h(x,n,m) - x_m b_{m-n}) \mod M$$
$$h(x,n+1,m) = (h(x,n,m) - x_n) \cdot \text{multinv}(b,M) \mod M$$

By using these rolling hashes, the algorithm efficiently identifies repeated sequences. It begins by calculating hashes for longer sequences and then contracts these windows, moving from the top down as illustrated in Fig. 2. The figure depicts a tree-like structure, where each arrow represents a contraction of the hash window, computed in $O(1)$ time. Hashes for shorter windows are only computed when necessary for later stages of the algorithm, reducing redundant calculations and enhancing efficiency. For our final complexity analysis, we focus on the number of hashes calculated for each window length i. Trivially, for each i, there are $N-i-1$ hashes to be computed. In the worst case, all substrings would require hash calculation, leading to a maximum complexity of $O(N^2)$ for the hashes, with N representing the length of the string under investigation.

To optimize our approach, we start with hashes with a maximum window length of $N/2$, as larger windows will inevitably result in overlapping segments, which cannot represent valid repetitions. By limiting the window size in this way, we ensure that each hash represents a distinct substring, maximizing efficiency in detecting unique repeating patterns.

Algorithm 1 Step 1: Hashing

```
prev_hash ← hashes
▷ Initialize prev_hash for the current
  iteration from previous one
for j = 0 to i − 2 do
        ▷ compute hashes for all substrings of length i
    hashes(j) ←
      (prev_hash − sub[i]) · multinv(b, M)  mod M
    hashes(i − 1) ← (prev_hash − sub[0] · b^(i−1))  mod M
     ▷ last element has to calculated by removing prefix
```

3.2 Selection

To identify repetitions, we compare hashes of a specific length. This process is best explained alongside the example in Fig. 3. We start by comparing adjacent hashes and, when matches occur, perform a full comparison to rule out hash collisions. Once matching hashes are confirmed, we greedily add them to the solution. This approach avoids a quadratic number of hash comparisons, as we skip overlapping or non-contiguous substrings.

In our implementation, this is managed by considering the length of the current substring i and comparing hashes separated by exactly i characters. This comparison must be performed i times to handle all possible offsets, represented on the vertical axis in Fig. 3. For each offset, the maximum number of hash comparisons is $\frac{N}{i}$.

As a result, the complexity of this step is $\frac{N}{i} \cdot i = N = O(N)$,

Authorized licensed use limited to: Centro Universitário Fei. Downloaded on June 23,2025 at 20:18:27 UTC from IEEE Xplore. Restrictions apply.

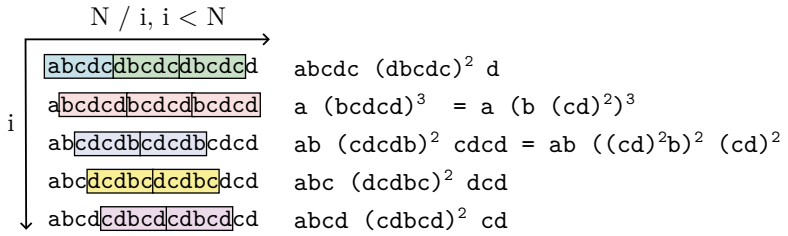


Figure 3 Example of Hash Comparison and Selection.

excluding the equality checks required to handle hash collisions. In the worst case, these equality checks could increase the total complexity to $O(N \cdot i)$.

It's important to note that different selection strategies can lead to varied results. The example in Fig. 3 demonstrates some of these possible cases. For performance, we implemented a greedy approach in our selection step, though an application-specific selection method could be substituted here. As long as the selection logic remains linear in time, it wont affect the overall complexity.

Algorithm 2 Step 2: Detecting Repetions

```
sequences ← []
counter ← 0
for all offset = 0 to i do
    for all elem_idx = 0 to |hashes| do
        if hashes(elem_idx · i + offset) =
          hashes((elem_idx − 1) · i + offset) then
            counter ← counter + 1      ▷ Repetition found
        else
            if counter > 0 then
                sequences ←
                  sequences + (elem_idx, counter)
                ▷ repetitions are represented by starting
                  index and number of repetitions
                counter ← 0                ▷ reset counter
    counter ← 0                            ▷ reset counter
```

Algorithm 3 Step 3: Selection

```
rep_selections ← []
last_selection =
  (last_selection_idx, last_selection_count) ←
  (−1, −1)
                    ▷ keep track last selected repetition
for all repetitions w = (elem_idx, count) in sequences
do
    if elem_idx ≤
      last_selection_idx + last_selection_count ∗ i then
          ▷ does w overlap with last selected repetition?
        Ignore w, due to overlap
    else
        rep_selections ← rep_selections + w
                              ▷ add to selected repetitions
```

3.3 Recursion

Following the selection step, two scenarios may arise: either repetitions are detected, or none are found. This recursive approach allows the detection of nested repetition structures by breaking down the string iteratively. If no repetitions are identified, the previous steps are repeated with a smaller substring length. However, when repetitions are found, we divide the result into segments that are either repeated or non-repeated. The algorithm is then recursively applied to these segments, with repetitions considered only once, as illustrated in Fig. 4. In the case of recursion, no additional smaller substring lengths i are considered, and the concatenation of the recursion results is returned as the output. The recursion has two base cases: for strings of length 1, the input is returned directly, and if the loop over all lengths down to 1 yields no detected repetitions, the input is also returned unaltered.

In the recursive step, we also pass previously calculated hashes to each subsequent call, allowing the algorithm to reuse existing computations and avoid redundant hashing. This optimization reduces the computational load in each recursive level, improving best case complexity.

This yields the full algorithm in Alg. 5. Since Step 1 is refering to hashes from previous iterations or recursive calls, this code calculates the first set of hashes. Since only the first of those must be generated by fully calculating the sum, while the rest can be calculated by deriving it from others, it generates an overhead in the order of $O(N)$. This is only neccessary once per execution of the complete algorithm.

To illustrate the algorithm as a whole, an example is provided in Fig. 5. The diagram depicts two iterations of the algorithm side by side. In the left column, the input is initially divided into overlapping subsequences of length N/2 and hashed. A scan for repetitions of length N/2 is then performed, but none are found in this example. As a result, the algorithm proceeds to the next iteration, visualized in the right column. The hashes for this iteration are derived from those computed in the previous iteration. Here, repetitions are detected and selected. Both the selected repetitions and non-selected subsequences undergo recursive analysis for further repetition detection (the details of these recursive calls are omitted from the visualization). Finally, the results are combined and returned.

978-3-8007-6515-7

@ VDE VERLAG GMBH · Berlin · Offenbach
Authorized licensed use limited to: Centro Universitário Fei. Downloaded on June 23,2025 at 20:18:27 UTC from IEEE Xplore. Restrictions apply.

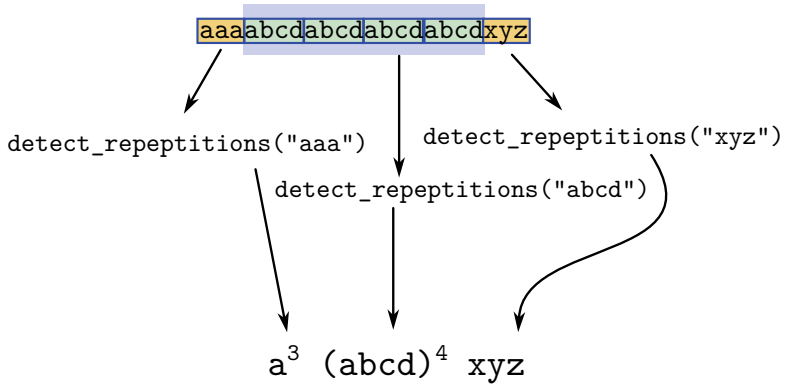


Figure 4 Recursive execution of the algorithm.

Algorithm 4 Step 4: Recursion

```
result ← [], prev_rep ← (−1,−1)
▷ rep_sections.first[0] is the starting index of
  first repetition selected

if rep_sections.first[0] ≠ 0 then
                    ▷ detected repetitions do not start at 0
    result ← [
      DETECT_REPETITIONS(
          s[0 : rep_sections[0] − 1], hashes)
    ]
for all repetitions rep = (start, rep_count)
  in rep_sections do
    if prev_rep ≠ (−1,−1) and
      prev_rep[0] + prev_rep[1] ∗ i < start then
                    ▷ there is a gap between repetitions
        result ← result +
            DETECT_REPETITIONS(
              s[prev_rep[0] + prev_rep[1] ∗ i : start − 1],
              hashes)
    result ← result + DETECT_REPETITIONS(
          s[start : start + i], hashes)
    prev_rep = rep
if rep_sections.last[0] + rep_sections.last[1] ≠ N then
                    ▷ detected repetitions do not end at N
    last_start ←
      rep_sections.last[0] + rep_sections.last[1] ∗ i
    result ← result +
        DETECT_REPETITIONS(
          s[last_start : N], hashes)
```

Algorithm 5 String Hashing and Repetition Detection

```
Require: String s with length N, Hashes rec_hashes from
         previous recursive steps
  if rec_hashes! = null then
      first_hash ← 0
      for i = 1 to N/2 do
          first_hash ← first_hash · b + s[i]   mod M
      hashes ← [first_hash]
      for i = 1 to N/2 do
          hashes(i) ←
            (hashes(i − 1) − s[i] · b^(N/2−2)) +
            s[i + N/2 − 1]   mod M
          ▷ Initialize hashes with
            all hashes of length N/2 − 1
  for i = N/2 to 1 do
        ▷ iterate over substring lengths starting from N/2
      Run Step 1: Hashing
      Run Step 2: Detecting Repetions
      if sequences = [] then          ▷ no repetition found
          continue                    ▷ no recursion needed
      Run Step 3: Selection
      Run Step 4: Recursion
      return result
```

978-3-8007-6515-7

@ VDE VERLAG GMBH · Berlin · Offenbach
Authorized licensed use limited to: Centro Universitário Fei. Downloaded on June 23,2025 at 20:18:27 UTC from IEEE Xplore. Restrictions apply.

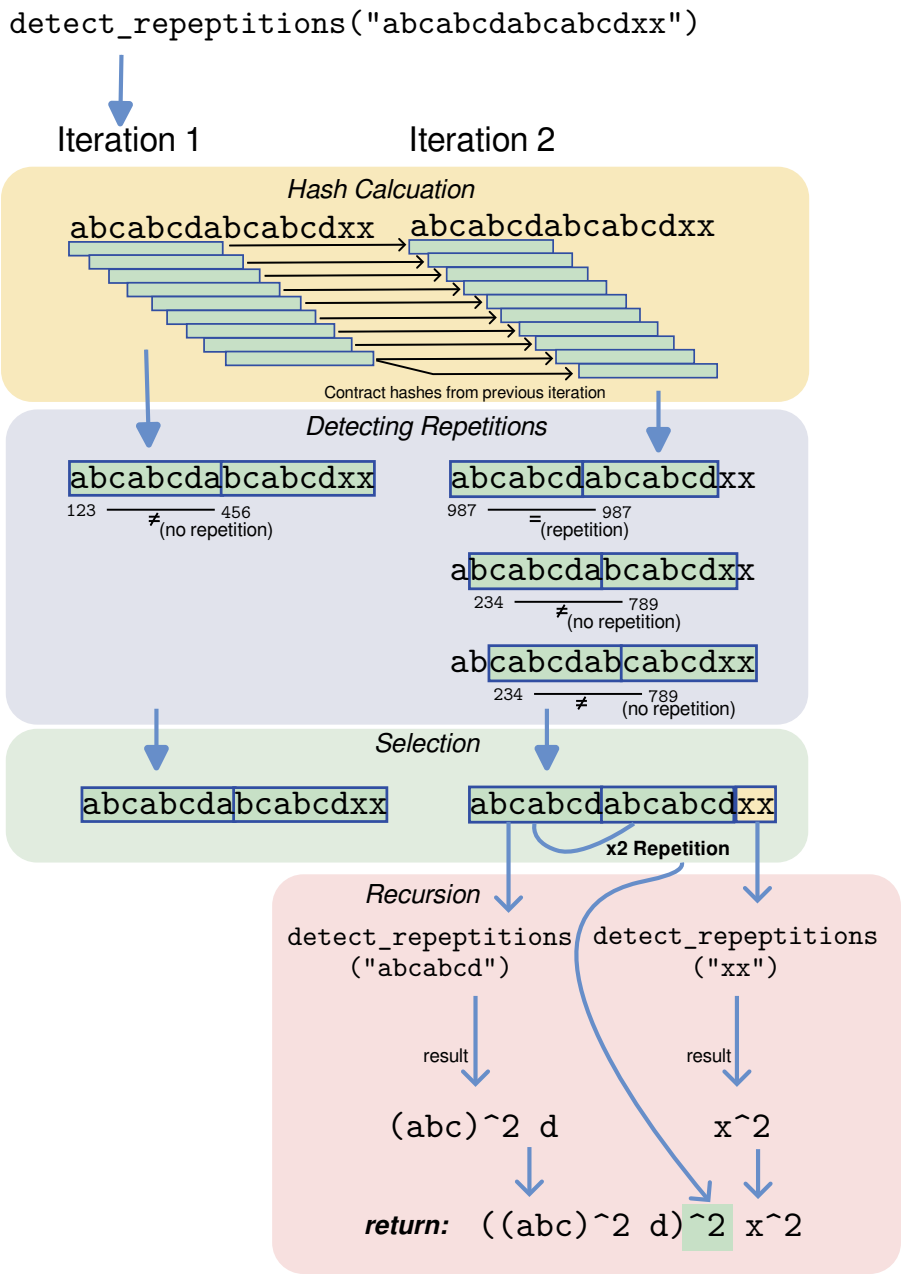


Figure 5 Example for an execution of the combined algorithm.

Authorized licensed use limited to: Centro Universitário Fei. Downloaded on June 23,2025 at 20:18:27 UTC from IEEE Xplore. Restrictions apply.

4 Evaluation

4.1 Complexity

An upper bound for the worst-case complexity can be easily established: The initial calculation of the hashes before the loop ammounts to $O(N)$. Step 1 calculates $O(N-i)$ hashes in constant time, leveraging rolling hashes. Step 2 performs $O(N)$ hash comparisons. Non-matching hashes are compared in $O(1)$ time, while matching hashes require $O(i)$ time for full comparison. Step 3, due to its greedy nature, takes at most $O(N)$. Step 4 makes, in the worst case, $O(N/i)$ recursive calls. However, when these calls are made, the outer loop exits, which is advantageous because hashes crossing section boundaries do not need to be recalculated or re-compared, and each repetition is only processed once. Thus, the actual worst case occurs when all iterations of the outer loop are performed.

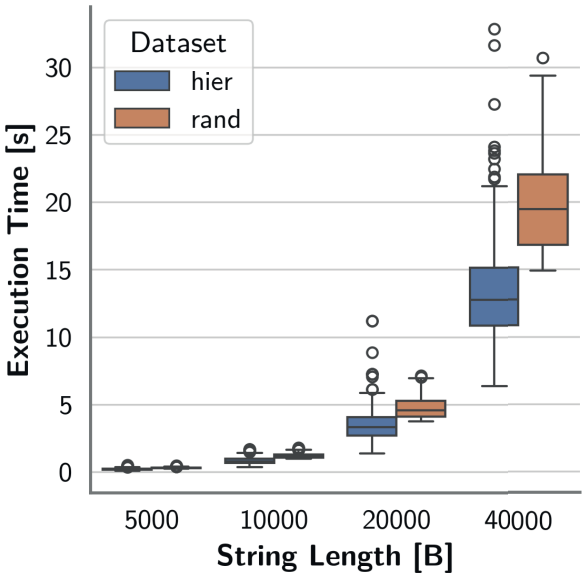
Summing up, this yields $O(N)+\sum_{i=1}^{N} O(N-i)+O(N)+O(N)+O(N/i) = O(N^2)$, ignoring non-matching hash comparisons. When including cases where comparisons are required for non-matching hashes, we get $O(N)+\sum_{i=1}^{N} O(N-i)+O(N\cdot i)+O(N)+O(N/i) = O(N^3)$. In practice, we anticipate much better performance. The best-case complexity occurs in a trivial case where a single character is repeated a power-of-two number of times, e.g., $\texttt{aaaaaaaa} = \texttt{a}^8$. For this case, each steps performance is as follows:

The initial calculation of the hashes before the loop ammounts to $O(N)$. Step 1 calculates 2 hashes in $O(N)$ time using rolling hashes. Step 2 performs 2 hash comparisons, which together take $O(N)$ for the comparison of the matching windows. Step 3 selects the single repeated sequence. Step 4 then recursively applies the previous steps to a substring of size $N/2$. This yields $O(N)+\sum_{i}^{\log_2 N} O(N/2^i) = O(N)$. In summary, the worst-case complexity of the algorithm is cubic, occurring only in a scenario with full hash collisions for all comparisons. However, in the best case, it operates in linear time. For the intended use cases, where repetitions are expected and hash collision are rare, it is unlikely that the worst-case complexity will significantly impact performance.

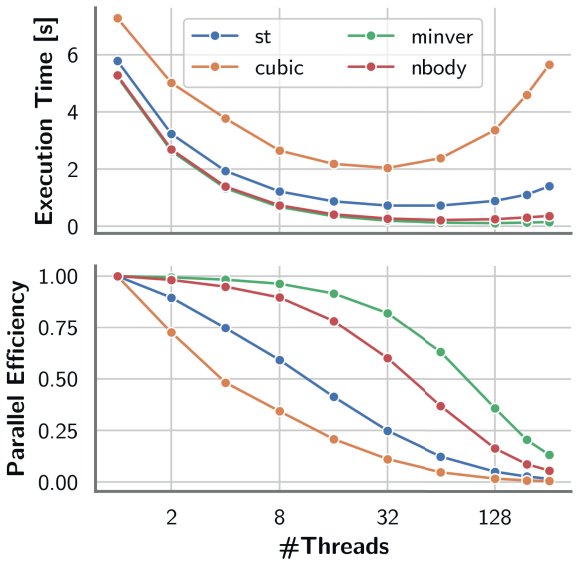
4.2 Experimental Results

We implemented the algorithm in Python and in C. The Python implementation was done with comprehesibilty in mind, while the C implementation is highly optimized. The code to these implementations is open source and can be found under `https://gitlab.uni-rostock.de/twinspace/detect-repetitions`.

For measurement purposes, we generated two datasets: one with purely random strings (*rand*) and another (*hier*) designed to contain nested repetitions. To evaluate the algorithms ability to detect nested repetitions, we created the hierarchical dataset using a recursive function, which builds byte sequences with layered repeating patterns. The function iteratively decides whether to append a flat (non-repeating) sequence or a nested sequence by controlling the recursion depth and repetition structure. For each nested segment, a smaller repeating pattern is generated and recursively embedded, resulting in sequences with multiple levels of repetitions. The specific algorithm and parameters controlling this process can be found in our open source implementation under `python/performance.py`.



(a) Single-Threaded Performance



(b) Muti-Threaded Performance

Figure 6 Performance of C implementation

In the following we will show results obtained with the C implmentation on a server system with two AMD EPYC 9654 CPUs. The box plot in Fig. 6(a) shows the obtained single-threaded performance:

We observed that the effects of parallelization were highly dependent on the input data. In our experiments, we parallelized the hashing step of the algorithm using OpenMP and obtained mixed results based on the input data and thread count. To illustrate these effects, we classified

Authorized licensed use limited to: Centro Universitário Fei. Downloaded on June 23,2025 at 20:18:27 UTC from IEEE Xplore. Restrictions apply.

floating-point instructions as introduced earlier, using the floating-point benchmarks from the Embench suite [13] (cubic, minver, nbody, st), processing the first 50000 entries with varying thread counts.
The results, shown in Fig. 6(b), reveal that for some datasets, such as cubic, performance degraded significantly beyond 32 threads. It is important to note, however, that there was always a speedup for some number of threads greater than 1. The optimal thread count and parallel efficiency (i.e., speedup per thread) varied across test cases. Notably, the example from the introduction achieved a parallel efficiency of approximately 0.6 with 192 threads, corresponding to the total number of physical cores available. As expected, increasing the number of threads from 192 (the physical core count) to 384 (the logical core count) resulted in the speedup increasing from approximately 121 to 153, albeit with reduced parallel efficiency, likely due to shared resources between logical cores in a simultaneous multithreading configuration. The data highlights two important aspects:

- The algorithm demonstrates sufficient efficiency for processing long sequences, especially with parallel processing. The example in the introduction, 10^6 items long, was processed in just 13.6 seconds.
- As anticipated, the presence of repetitions in the input data results in improved performance, underscoring the algorithms effectiveness in leveraging repetitive patterns to reduce computational workload.

5 Conclusion

In conclusion, the proposed method effectively addresses the challenges of processing repetitive microarchitectural sequences by detecting and representing hierarchical structures within these data streams. This capability is particularly beneficial for machine learning applications in domains like power consumption prediction, where such repetitive structures can lead to inefficiencies in both model training and inference phases. By accurately identifying nested and n-fold repetitions, the algorithm reduces the need to process redundant sequences, achieving a more compact representation that captures the essential characteristics of the data.

5.1 Future Work

The current algorithm demonstrates strong performance, yet there is substantial potential for improving the parallelization of the workload. The characteristics of the input data should be considered and the other potentials, especiallty the parelleization of the recursive calls might also improve the overall performance. Through profiling we found that most of work is spent in calculating hashes. This step is very uniform and lends itself to GPU acceleration. This venue should be explored, since the degree of paralellism is very high at certain points of the execution. Future work will focus on addressing these limitations to unlock the full parallel potential of the algorithm, exploring more sophisticated parallel strategies to maximize scalablility.

Acknowlegement

We acknowledge support by the Federal Ministry for Economic Affairs and Climate Action (BMWK) due to an enactment of the German Bundestag under Grant No. 01MN23013H.

6 Literature

[1] T. Hönig, B. Herzog, and W. Schröder-Preikschat, “Energy-demand estimation of embedded devices using deep artificial neural networks,” in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 617624.

[2] F. Fricke, S. Scharoba, S. Rachuj, A. Konopik, F. Kluge, G. Hofstetter, and M. Reichenbach, “Application runtime estimation for aurix embedded mcu using deep learning,” in *Embedded Computer Systems: Architectures, Modeling, and Simulation: 22nd International Conference, SAMOS 2022, Samos, Greece, July 37, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 235249.

[3] D. E. Knuth, “Structured programming with go to statements,” *ACM Comput. Surv.*, vol. 6, no. 4, p. 261301, Dec. 1974.

[4] From noise to clarity: Addressing data redundancy in ML. [Online]. Available: https://stelar-project.eu/data-redundancy-in-machine-learning/

[5] D. Gusfield and J. Stoye, “Linear time algorithms for finding and representing all the tandem repeats in a string,” *Journal of Computer and System Sciences*, vol. 69, no. 4, pp. 525–546, 2004.

[6] A. Apostolico and F. Preparata, “Optimal off-line detection of repetitions in a string,” *Theoretical Computer Science*, vol. 22, no. 3, pp. 297–315, 1983.

[7] M. G. Main and R. J. Lorentz, “An o(n log n) algorithm for finding all repetitions in a string,” *Journal of Algorithms*, vol. 5, no. 3, pp. 422–432, 1984.

[8] A. Nakamura, T. Saito, I. Takigawa, M. Kudo, and H. Mamitsuka, “Fast algorithms for finding a minimum repetition representation of strings and trees,” *Discrete Applied Mathematics*, vol. 161, no. 10, pp. 1556–1575, 2013.

[9] M. Burtscher, “VPC3: a fast and effective trace-compression algorithm,” *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, pp. 167–176, Jun. 2004.

[10] ——, “TCgen 2.0: a tool to automatically generate lossless trace compressors,” *SIGARCH Comput. Archit. News*, vol. 34, no. 3, pp. 1–8, Jun. 2006.

[11] “The VPC trace-compression algorithms.” [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1514414

Authorized licensed use limited to: Centro Universitário Fei. Downloaded on June 23,2025 at 20:18:27 UTC from IEEE Xplore. Restrictions apply.

[12] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms,” *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[13] “Embench repository.” [Online]. Available: https://github.com/embench/embench-iot

Authorized licensed use limited to: Centro Universitário Fei. Downloaded on June 23,2025 at 20:18:27 UTC from IEEE Xplore. Restrictions apply.