École Polytechnique Fédérale de Lausanne

Scalable Computing Systems Laboratory
Optional semester project

Machine Learning for Page Prefetching

Victor Garvalov

Supervisors : Rafael Pires, Martijn de Vos
Professor : Anne-Marie Kermarrec

Code repository
Data Drive
January 11, 2025

Abstract

As workloads become more and more memory-hungry, classical memory management techniques such as to-disk swapping fail to scale. To circumvent this, research has proposed several different layers of memory management optimizations throughout the years. With the introduction of such different systems, custom memory management mechanism are put in place to make best use of their technology. Namely, this makes it difficult to find out to what extent certain prefetching policies are effective. In this work, we aim to build a technology-agnostic prefetcher which would work from the OS Kernel level. To leverage all information at the kernel's disposal, we gather page fault traces for a set of PARSEC benchmarks. The traces we gather are composed of several new features not common in typical memory access traces used for building cache prefetchers. Notably our traces contain the Program Counter (PC, sometimes present in memory access traces), the stacktrace, the trapframe registers at fault time, and optionally, the surrounding assembly instructions of the running code. We then perform analysis on these traces focusing of the stacktrace feature. We propose two new ways to visualize the page fault traces. Building on the gathered insights, we propose a slightly different formulation for the prefetching problem. This formulation would facilitate the task of predicting the next faulty address at the expense of a new task, predicting the next path in code or PC which would generate a fault. We therefore propose an alternative system with multiple prefetchers, one per execution path/PC which has generated a fault in our process. We run various experiments in this context, using Leap[1] as a baseline for comparison, as well as NLinear [2], a linear ML model optimized for time-series forecasting. We propose two new metrics (Success and Precision@K) to perform trace-based page prefetcher evaluation, without needing to simulate a complex OS with different layers of memory. Our results are inconclusive. We find that in some cases, our approach outperforms the basic Leap algorithm, reaching at its peak a 25% success and 23% precision improvements over base Leap for the fluidanimate benchmark restricted to 25% of its memory requirements. In other cases, such as for the streamcluster, canneal and dedup benchmarks, our approach is worse at predicting future page faults. NLinear on the other hand consistently performs worse than basic Leap for page prediction. We also evaluate the latter for the newly-created task of "future group prediction" (i.e.: predict what the next path/PC generating a pagefault will be). It achieves 50.2% and 48% Average Precision@10 for predicting a PC and stacktrace ID respectively.

Unfortunately, due to an overlook on our side explained in Section 3.1, all our work is potentially meaningless. Further tests will be required to verify this.

Table of contents

1 Introduction

Since the early days of computer science, memory has been identified as being one of the main bottlenecks in program executions [3–5]. Although the usage of swap devices is a reliable solution to make up for latencies induced by faults in DRAM in personal computers, more demanding workloads are usually run in data centers, making those servers require massive amounts of memory. This in turn increases the amount of data copying between the different stacks of memory, which doesn't scale for simple memory management techniques such as to-HDD swapping. To address this, the memory stack was extended over time, introducing new layers through:

- technological advancements, such as Non-Volatile Memory [6] and SSDs [7], or more recently CXL [8]
- shared memory, across CPUs through Non-Uniform Memory Access (NUMA) nodes [9], or over the network through disaggregated memory (RDMA) [1; 10; 11]

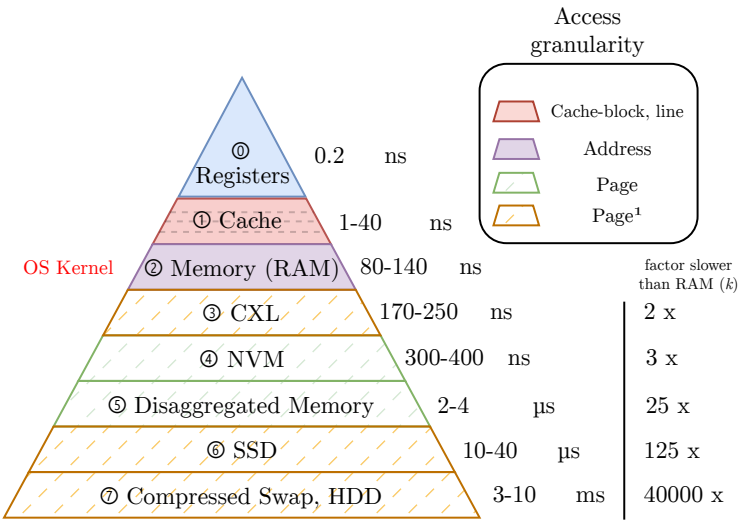


Figure 1: Modern memory hierarchy with each layer's access latencies [8]

Figure 1 shows the different layers potentially present in a modern memory stack, ordered by access latency. For simplicity, we will refer to layers of the memory stack in the following sections by the circled number next to them (e.g.: ①).
From the point of view of the OS in charge of memory virtualization, when an application requests data that is missing in its local memory, the data will be fetched from one of the underlying layers. We will refer to such failed accessed as **page faults** (although due to implementation details surrounding CXL and RDMA, some of those faults are not *major* page faults as commonly referred to in the OS community).
In memory pressured scenarios where RAM is used close to its capacity, this can thus lead to big delays, as from the point of view of the application each faulty access will be a factor k slower than usual. In the worst case, where there are n such subsequent accesses which aren't

[1]Since these orange layers rely on underlying hardware, access is done at the granularity of device-specific sector sizes. From the OS' point of view however, up until driver code, the access granularity is on a per page basis.

batched, this will result in a data path that is k^n exponentially slower.

This "miss" concept is not restricted to the RAM layer. In practice, two common techniques exist to alleviate it and are used in parallel. The first one, the *eviction mechanism*, consists of choosing which data will remain in a certain memory layer, given an access trace. It aims to predict which of the currently held data is more likely to be useful in the future. Better eviction mechanism can therefore reduces misses by holding the relevant data for longer. The second one, the *prefetching mechanism*, consists of choosing which data will be brought in (from a slower memory layer), given an access trace. It aims to predict which of the currently un-held data is more likely to be used in the future. Better prefetching mechanism can therefore reduce misses by preemptively bringing-in (prefetching) future relevant data.

With the introduction of the different layers, many custom eviction and prefetching policies have been put in place. This makes it difficult to point out exactly whether the policy itself was effective, or whether the system's design as a whole was helping it. In this work, we will focus on prefetching policies. We place ourselves as the OS at the local memory level (after (2)) and before all of {(3),...,(7)}), and are interested in all application-allocated data for a 4KB-page system. To remain consistent with the information available in current OSes, we consider only page-fault granularity traces (i.e.: we do not have every memory access but only those done to pages which were not resident to local RAM (2)). We aim to build a prefetcher agnostic to the underlying memory-system set up past (2), with the end goal of reducing all page faults.

As we will see in the next section, prefetching has notably extensively been studied in the domain of caches. However, there are two key differences to take into account when comparing the tasks of cache, and page prefetching.

1. Caches' prefetchers must be less complex
    a. As they are embedded inside the cache itself, more complex cache prefetchers will require more physical circuit space in their "implementation". Due to the limited size of the CPU, this would have to come at the expense of a reduction in either the cache size itself, or one other of the CPU components, which is in general not desirable.
    b. As seen in fig. 1, caches operate on a fraction of RAM's latency. A prefetcher of higher complexity will thus be restrained in the amount of "thinking" it can do before prefetching becomes useless.
2. Apart from the data they serve and potentially Program Counters (PCs), caches don't have access to higher order information such as which process made which access, or what the current execution status of the process is.

As such, we will attempt to understand what characterizes workloads as great for page prefetching. We aim to exploit those two key differences by experimenting with various prefetching techniques. Namely, we leverage some of th eextra information, noteably the stacktrace, through machine learning, and build relevant datasets for the task of page prefetching. Section 2 will cover some important background surrounding the subject, Section 3 will describe the data we have gathered, as well as present some basic analysis surrounding it. Finally, Section 4 will formalize the problem of pretching in our context, propose an alternative version which makes use of the stacktrace, and report the results of some base experiments done, which can be used as reference for future work.

2 Background and Related Work

2.1 Prefetching

Many systems exist for cache prefetching [12], some offering insights on the usage of machine learning for the task at hand. In this section, we will first describe some basic cache prefetchers which could be used as inspiration for simple heuristic page prefetchers, then look at ML-driven prefetchers, before presenting the few existing page prefetchers.
We consider addr_t as the faulty addresses at time t, and page_t the corresponding faulty page number (i.e.: $\text{page}_t = (\text{addr}_t \,\&\, \sim\texttt{0xfff}) >> 12$ for a system using 4KB pages).
Table 1 summarize a selected few cache prefetchers, while we give a more thorough description hereunder.

2.1.1 Cache Prefetchers

Cache prefetchers operate either in between the individual level of caches (e.g.: L1↔L2, ①), or between the Last Level of Cache (LLC) and RAM (①↔②). We categorize them in 3 types:

- ✧ **Stride prefetchers**. Many stride prefetchers exist, each ever so slightly different than the other. The base idea was introduced in 1991 [13] and can be summarized by a 3-state machine. It tries, for each tag (address), to compute a stride, or delta: $\Delta_t = \text{addr}_t - \text{addr}_{t-1}$. It then predicts and prefetches $\text{addr}_{t+1} = \text{addr}_t + \Delta_t$ updating its success state as feedback is received from the lower levels of cache. This idea is still optimized for accuracy [14] and resource consumption to this day [15]. Stride prefetchers were then generalized to be able to predict multiple, interleaved, delta streams heuristically to some extent [16; 17].

- ✧ **Correlation prefetchers**. Markov [18] and correlation [19; 20] prefetechers treat address accesses as streams, and probabilitstically prefetch new addresses using a Markov model, where the prefetched address is such that $P(\text{addr}_{t+1}|\text{addr}_t, \text{addr}_{t-1}, \ldots)$ is maximised. They are able to predict more complex patterns, like indirect accesses or pointer fetchers, but are however more expensive to build to reach their theoretical values, and thus not used in practice [21].

- ✧ **Compiler-aided prefetchers**. An alternative to hardware-issued prefetches are compiler, or software-hinted prefetches [22; 23] (e.g.: through ISA-support like x86's `PREFETCH`h instruction). Unfortunately, without runtime information, it is difficult to know which prefetch is effective, thus making them not used in practice either.

As shown in Table 1, we can also classify all prefetchers by the memory access pattern [24] they have been optimized for.
Finally, while most [14; 15; 20; 21; 25; 26] of the other aforementionned works, use the Program Counter as an indicator for spatial locality (by using it as an index in their history tables), some [16; 17] argue that in the context of caches, a PC-granularity breakdown breaks the temporality in the access patterns, and thus choose to not include it in their systems. These works also relevantly point-out that most commercial CPUs do not have PC information on the cache-level, thus inciting for PC-less designs. As noted in difference 2., this is not an issue for page prefetching.

2.1.2 ML for cache prefetching

Recent advances in the field of Machine Learning have motivated systems researchers to turn to ML for the task of cache prefetching. Several works of interest stand out:

Pattern	Relation	Example	Prefetch	Relevant Work
Constant, Delta	$A_n = A_{n-1} + d$	`*ptr`, streaming, array traversal, `a[i], a[i][j]`	`a[i+d][j]`	[13–17; 23]
Pointer Chase	$A_n = Ld(A_{n-1})$	`next = current->next`	`current->next->next`	[22; 23]
Indirect Delta	$A_n = Ld(B_{n-1} + d)$	`*(M[i])`	`*(M[i+d])`	[19]
Indirect Index	$A_n = Ld(B_{n-1+c} + d)$	`M[N[i]]`	`M[N[i+d]]`	[18]
Constant + offset reference	$A_n = B_n + c_1$ $B_n = Ld(B_{n-1} + c_2)$	Linked list traversal	Next struct in the list	[22; 23]

Table 1: Memory prefetchers usefulness on different patterns

- *Hashemi et al.* [27] establish a modern ML prefetching baseline by using a Long Short Term Memory (LSTM) as base model. They take as input address deltas ($\Delta_i := \text{addr}_t - \text{addr}_{t-1}$) as well PCs, cluster them by similarity, and use an embedding of their concatenation as input. They predict top K=10 next deltas and use only the N=50.000 most frequent deltas as input vocabulary.

- *Srivastava et al.* [28] also use an LSTM, taking only deltas as inputs, and predicting only $K = 1$ next delta. In addition, they achieve very low memory overhead by using a bit vocabulary and compressing deltas down to 16 bits.

By predicting deltas, it is intuitively clear that this approach models strided patterns well. Furthermore, it can also be argued (though it has not been shown) that it can be applicable to more complex patterns to an extent. A piece of program performing a linked-list traversal, or pointer chasing, can be interpreted as simply jumping from one address to another (which can be placed anywhere in memory, independently of the former address). This simply shifts the base offset of the prefetcher, and turns the problem into the task of finding a $\Delta_{t'}$ at page fault time t which would satisfy $\text{addr}_t = \text{addr}_{t-1} + \Delta_{t'}$. $\Delta_{t'}$ would somehow have to be determined by the prefetcher based on the past, and the complexity of this task is the same as the one of predicting the address directly.

- *Shi et al.* [29] propose *Voyager* – a hierarchical architecture taking as input PCs and addresses. Unlike the previous two authors however, they split the address into page offset (lower 12 bits when using 4K pages) and number (the rest of the address, page_t). They then compute each of those features' embeddings before using them as input to two LSTMs - one predicting the prefetched address page number, the other its page offset.

Having different embeddings (and predictors) for the page offset and number gives more expressiveness to the model, as it tries to capture different aspects of the access pattern.

- *Wu et al.* [30] use a Hebbian networks (whose architercture is inspired from the human brain) to build a resource-efficient alternative to [27]. They achieve similar performance for while having only $\frac{1}{3}$ of the weights, and using only integer operations (no floating point ops).

All four models above predict *addresses* and prefetch from RAM into the LLC (i.e.: they stand between (1) and (2)). Furthermore, by their nature, they can also track patterns when accesses cross page boundaries, an issue first identified by [17] which affects most previously mentioned general cache prefetchers.

- Finally, *Zhang et al.* use transformers [31] to predict (caches' ≈ 8-bit) block indices, within a page offset based on global history. Their input thus comes after the L2 cache (①), and is used to prefetch from RAM into the LLC (②→①), making it a hybrid between the previous works. Predicting such a small amount of bits, they are able to simply use a binary vocabulary and feed the block index to the model without needing to use tokenizers.

While the task of cache block prediction is slightly unusual, it is the first to our knowledge to use transformers [32] for prefetching, being a motivation for the work presented in section 2.3.

2.1.3 Page prefetchers

Disclaimer: the works mentioned in this first paragraph have been found while writing the final report. Notably, section 4 ignored their existence and did thus not use them as additional baselines for comparison.
Since Linux 2.6, a sequential page prefetching mechanism called readahead [33] is implemented to prefetch both file-backed and anonymous pages from disk. It works by simply prefetching a window of K next, closest, pages when a page page_t faults (i.e.: it fetches $\text{page}_t + 1, \text{page}_t + 2, \ldots, \text{page}_t + K$). Many other mechanisms have been developed to propose alternatives to readahead. They notably make use of correlation analysis, or advanced stride detection, similarly to the works mentionned in section 2.1.1 [34–38]. As they are mainly interested in file-backed page fetches, these solutions are exclusively placed between ② and one of ⑥ or ⑦, ignoring the existence of previous layers.

Leap [1] is page prefetchers implemented in the context of RDMA (thus placed between ① and ⑤) which noteably considers all page faults. It improves over basic Linux prefetching, which is why we later select it as a baseline for comparison. Leap works as a strided delta prefetcher using a simple Boyer-Moore majority vote algorithm over the stream of page requests. For instance, consider the following hypothethical stream of w faulting addresses (where the current faulting page addrt_t is in red) that Leap will see as input:

`0x1000` `0x3000` `0x5000` `0x9000` `0xb000`

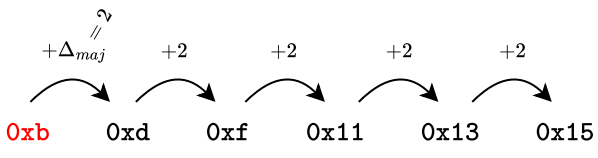
and their corresponding faulting pages numbers:

`0x1` `0x3` `0x5` `0x9` `0xb`

Leap will first compute the page deltas $\Delta_i = \text{page}_i - \text{page}_{i-1}$:

2 2 4 2

Then, it applies a majority vote algorithm to find the most frequent delta Δ_{maj} – here $\Delta_{maj} = 2$ as 2 occurs 3 times compared to 4 which occurs only 1 time.
Finally, it prefetches a window of K future predicted accesses off the faulty page according to the found Δ_{maj}, s.t. the predicted $\widehat{\text{page}}_{t+i} = \text{page}_t + i \cdot \Delta_{maj}\ \forall i \in \{1, \ldots, K\}$. In this instance, assuming $K = 5$, leap (pre)fetches pages:



Note that Leap only prefetches when it is *certain enough* of its prefetch, which it it considers to happen whenever the majority delta Δ_{maj} occurs more $\geq \frac{w}{2} + 1$, i.e.: if more than half the the window exhibits a Δ_{maj} pattern.

Finally, Canvas [39] is another recent Disaggregated Memory system. It proposes an alternative to Leap's heuristic prefetcher based on application semantics taken directly at runtime through object tracking in the JVM whenever Kernel prefetching (through `readahead` or Leap) continuously fails.

2.2 Time-Series forecasting

To be able to perform accurate ML page prefetching, it is important to model the task correctly. While the previous work described in section 2.1.2 intuitively uses LSTMs and/or transformers because of their capabilities to capture long term temporal dependencies and patterns, in this section we take a peak at recent work in the field of ML for Long sequence time-series forecasting (LSTF).
Zhou et al. [40] point out the inability for LSTMs to perform accurately at the LSTF task, and the slowness of typical transformer models induced by the autoregressive aspect. To circumvent this, they propose an efficient implementation of transformer models through Informers. Informers outperform most previous models used for Time Series Forecasting, including LSTMs. Several works build on top of Informers[41; 42]. Of note are *Kim et al.* [43] identify that some non-stationary information (such as the time-varying mean and variance) is lost when using transformers, even in simpler cases of seasonal data. They restore it by providing a data normalization technique, RevIN, which achieves up to 2x improvement on Nasdaq stock exchange prediction. In parallel, *Zhou et al.* [44] provide an even more efficient implementation of transformer models. It can be trained with linear complexity w.r.t. the sequence length, compared to the standard transformer's quadratic complexity, and even outperforms some of the previous models.
Finally, *Zeng et al.* [2] question the need for complex models altogether. By using one simple linear layer, the NLinear model, they manage to outperform all transformer architectures, at a fraction of the cost. They achieve their results with only $\approx 140K$ parameters, thus yielding an inference time of $\approx 0.4ms$ for 96 future predictions.

2.3 Previous and ongoing project iterations

This work is a continuation of an ongoing collaboration with Alexandra (Sasha) Fedorova, UBC. Her lab currently focuses on testing and adapting *Hashemi et al.* [27]'s LSTM for our context of page prefetching. In a previous iteration of the project, we have explored using a standard transformer [31] to perform prefetching and have found that, unsurprisingly, performance was application-dependent. We achieved good ($\approx 85\%$) accuracy for the `fluidanimate` benchmark, yet very low accuracy ($< 0.1\%$) on `canneal`.

As introduced in Section 1, this work will try to understand what makes such dissimilarities possible by digging into the data available, and then try to learn from the gathered insights to improve on the previous work.

3 Data

3.1 Dataset Gathering

To analyse applications' behavior, we need to gather traces representative of what an OS dealing with Figure 1's complex memory hierarchy would see. That is, we want traces of page faults. These traces should ensure that they are not polluted by any pre-existing prefetch policies, such as Leap or Linux's `readahead`.
Currently, five main options exists to gather information about application memory accesses:

1. CPU counters (e.g.: PEBS [45]) through tools like `perf` [46]
2. Full memory access traces through binary emulation tools like `Pin` [47]
3. Kernel probing through Kprobes with tools like `SystemTap` [48]
4. "Live" Kernel logging to, e.g., `/var/log/` through custom kernels
5. Userspace logging through appropriate usage of `userfaultfd` [49]

Table 2 summarizes the key differences between the different methods.

Method	Usage Complexity	Full Trace	Granularity	Features
CPU Counters	Low	✗[2]	Page Fault or Memory Access	PC, Fault address
Binary Emulation	Low	✓✓	Memory Access	PC, Fault address, stacktrace, registers
KProbes	Medium	✓	Page Fault	PC, Fault address, with more work, registers
Kernel Logging	High	✓	Page Fault	PC, Fault address, stacktrace, registers
userfaultfd	Medium	✓	Page Fault	PC, Fault address, stacktrace, registers

Table 2: Memory tracing tool characteristics

To obtain page fault granularity traces from Memory Access streams, one would need to simulate a live system (kernel, and underlying memory stack), which is infeasible in practice. This makes the `userfaultfd` option most viable. We thus use a custom fork of `fltrace` [50], a wrapper around `userfaultfd` which redirects all page faults to userspace for statistic collection.

While writing this report, we notice the following lines in the `userfaultfd` man page: "*To register with userfaultfd minor fault mode, the user needs to initiate the* `UFFDIO_REGISTER` *ioctl with mode* `UFFD_REGISTER_MODE_MINOR` *set.*", while our work thought that, by design, `userfaultfd` also caputerd minor page faults. Upon looking at `fltrace`'s usefaultfd initialization code, we notice that this flag seems to not be set. We have contacted Anil Yelam, author of the tool, who ascertains that minor faults **are** captured. In the event they are not, it is possible that Linux' readahead prefetcher interfered with our traces, thus rendering our work meaningless. Because of the deadline surrounding the report, we have not had the time to verify and test this ourselves.

[2]To get every page fault, perf must be configured to capture at granularity 1. This will initialize a CPU microcode callback request at every page fault event. These events are stored on limited-size CPU ring buffers. Since the callbacks are done asynchronously, events and their data (e.g.: which page faulted) get overwritten in the case of high memory pressure, leading to data losses of up to $\approx \frac{2}{3}$ of a full trace.

Out of the box, the following features are captured for each page fault:

- the memory address of the faulty page
- the stacktrace – a list of PCs revealing the function call-stack which led to the page fault (similarly to what would be thrown by a debugger in the case of an exception)
- the flags of the page fault (defining whether the access was a page Read, Write, or Write-Protect in the case of newly allocated page)

Through a custom kernel patch, we extend the tool to also capture

- the accurate PC of the traced program, which doesn't always correspond to the last element of the stacktrace (e.g.: because of function inlining, calls into the standard library, or `fltrace` noise)
- the registers of the trapframe

Given the application's disassembled executable, as well as those of the shared libraries it accesses, we can also get, in post-processing, a window of previous and future executed assembly instructions. We provide a script to do this in our code repository. When used to extend the trace, one data point, corresponding to one page fault at addr_t, in any of the traces X can thus be described by :

$$X[t] = \{ \begin{array}{c} \overset{\text{addr}_t}{0\text{x}7\text{fffd}102\text{a}000}, \\ \overset{\text{PC}}{0\text{x}555555558999}, \\ \overset{\texttt{flags bitmap}}{0001}, \\ \overset{\text{stacktrace}}{[0\text{x}555555558999, \ldots, 0\text{x}7\text{ffff}4326850]}, \\ \text{neighbor instructions, in red, faulting instruction} \\ \texttt{add \%rdx,\%rax; mov \%eax,-0x10(\%rbp);mov 0x8(\%rax),\%eax} \end{array} \} \equiv \{ \begin{array}{c} \alpha, \\ \beta, \\ \gamma, \\ \theta, \\ \phi \end{array} \}$$

We gather traces for various benchmarks of the PARSEC suite [51], used namely in [28]. Table 3 summarizes our choice.

Name	Description
bodytrack	Body tracking of a person
canneal	Simulated cache-aware annealing to optimize routing cost of a chip design
dedup	Next-generation compression with data deduplication
facesim	Simulates the motions of a human face
ferret	Content similarity search server
fluidanimate	Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method
raytrace	Real-time raytracing
streamcluster	Online clustering of an input stream

Table 3: Traced benchmarks

As we are interested in page faults, we impose memory restrictions (through fltrace's `-L` and `-M` flags) on the applications we trace to simulate a memory-pressured system . We thus collect

5 different granularities of traces corresponding to {20%[3], 25%, 50%, 75%, and 100%} of the application's total memory usage (i.e. in a 75% trace, the application is allowed only up to 75% of its maximum used memory – the lower the number, the more memory pressured the system is, making our trace more granular). The 100% trace thus completely fits in memory - all page faults it contains are just *cold faults* – first-time accesses to pages which cause a page fault to actually allocate the page.

We computed total used memory in two different ways, through the `/usr/bin/time` tool and through the Valgrind `massif` tool. We eventually used `massif` as it provides slightly more consistent results (over 100 runs of the same application, it always computed the same value, whereas '/usr/bin/time' had the very small standard error of 0.00523). The difference between the tools was statistically insignificant however ($p<0.0001$), so both can be used equivalently as ground truth for application memory estimation. We tried getting traces with even bigger memory pressures (10% and below), however most traced programs crashed mid-trace.
Note that as shows Figure 2, the number of page faults does not necessarily grow linearly with memory pressure. While this result in itself is hard to quantify as intriguing or not, we find that the negligible increase in trace size for the 50% and 75% scenarios is not normal and further discuss why that could be in section 5.
Of specific interest are:

1. `streamcluster` which has the lowest memory requirements and thus is difficult to simulate memory pressure for (the number of page faults remains the same across all percentages, and similarly for `dedup`), and

2. `canneal`, which increases its trace size by a factor of $> 300\times$ at a granularity of 25% already. This already yields a trace of $\approx$ 3GB with 11+ million data points, which is skip getting a 20% trace for it.

Unless otherwise specified, our analysis and ML work will be done using the 25% traces, as we found them to be a good middle point between having enough data to get interesting insights, and not being too long to run experiments on.

We provide a step-by-step guide to install the custom kernel, recreate those traces, or build new ones in our code repository. The raw data is available here as `.csv` files, where all features are encoded as hex strings. In the following sections, we use preprocessed versions of the traces that we obtain by:

1. Removing rows (page faults) whose PC $\in$ {`0xffffffffb2987dba`, `0xffffffffb2987d81`, `0xffffffffb2105a48`, `0xffffffffb1cff9e4` } (modulo ASLR), as they point to neither the executable nor some library referenced by it.

2. Removing rows (page faults) caused by `fltrace` itself

3. From the stacktraces, removing all functions which are called after going through `fltrace` code[4]

[3]We added this granularity towards the middle of the projects. The analysis work of section 3.2 was done before we gathered it.

[4]This is done because of `fltrace`'s design which preloads itself into the traced application binary as a library and smartly uses signals to get a stacktrace from a non-faulting thread at page fault time. The signal-calling PC (which is part of `fltrace` but is executed as if it came from the traced application) is thus part of the stacktrace, hence why we remove it.

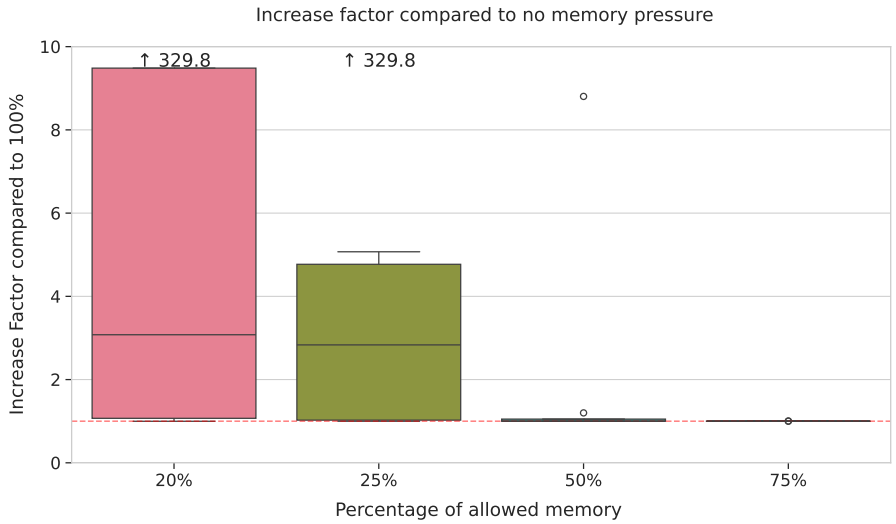
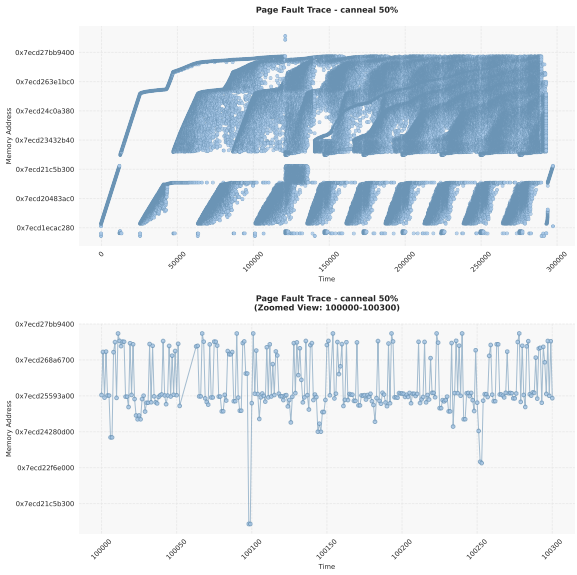


Figure 2: Boxplot aggregating, for all benchmarks, trace size increase compared to the no memory pressure scenario. A factor of k for a given percentage of allowed memory $X\%$ means that when left at $X\%$ of its total memory, there are $k\times$ more page faults when running the application.

3.2 Data Analysis - exploiting the stacktrace

To contextualize the task of page prefetching, let us start by looking at two raw page fault traces.



(a) Canneal page fault trace, restricted to 50% of memory

(b) Facesim page fault trace, restricted to 25% of memory

Figure 3: Raw page fault traces (top) and their zoomed-in versions at timestep 100000 (bottom)

Figure 3 shows the difficulty of the task of page prefetching. Firstly, while we can visually identify some patterns, it is unclear of their periodicity, and to what extent the surrounding noise can prevent systems from identifying them. Secondly, while it seems that there are many data points per timestep, when zooming in on a portion of the trace, we can see that the workloads are simply very volatile, jumping from one address to a distant one very quickly back and forth.

To explain this volatility, consider the toy program in fig. 4a, which performs a linear array traversal (delta pattern from Table 1) on two arrays. The loop will generate the following access stream: `0xA1000`, `0xB1000`, `0xA2000`, `0xB2000`.
This simple yet not uncommon example (many programs access more than one array/data stream in a loop iteration) easily showcases the jumpy behavior seen in our traces. Even for caches however, the more advances stride prefetchers described in section 2.1.1 can easily unwrap the two data streams by looking at the program counter which generated the fault. Modern programs are however very modular [52], and delegating memory accesses is not uncommon, as showcased in fig. 4b. This latter loop will produce the following access stream at PC Z which appears complicated to predict: `0xA1000`, `0xB6000`, `0xA2000`, `0xA3000`, `0xB8000`. However in reality, the accesses' true source can be seen as the call point in the loop. Since this information is available through the stacktrace, one can smartly group the accesses in the following way: (PC X, PC Z) $\rightarrow$ (`0xA1000`, `0xA2000`, `0xA3000`) and (PC Y, PC Z) $\rightarrow$ (`0xB6000`, `0xB8000`), which both are learnable delta patterns.

```
for(int i = 0; i < 2;i++){
    ret += A[i]
    ret += B[i]
}

```

(a) Multi-array looped access

```
for(int i = 1; i < 4;i++){
    ret = get_and_log(A,i);        //PC = X
    if (i%2==1){
        ret+=get_and_log(B,i+5); //PC = Y
    }
}
int get_and_log(arr,i){
    //do some logging
    return arr[i];                 //PC = Z
}

```

(b) Delegated access

Figure 4: Motivational toy examples. Consider that both arrays hold elements of size 4KB (1 page), and that `A` is located at `0xA0000`, while `B` at `0xB0000`. Consider that every array access is faulty.

As such, we propose analyzing traces on a stacktrace-granularity rather than PC-granularity as is done in previous works. We call this method **grouping by stacktrace**, and contrast it to **grouping by PC**.

3.2.1 Execution graphs

To further motivate our choice, we abstract away the time dimension from our traces and visualize them using **execution graphs**. One such directed graph can be seen in Figure 5. Nodes in our graph correspond to Program Counters at various levels down the stacktrace, and are labeled by the Program Counter's address (position inside the code). Green nodes correspond to places inside the traced application's binary, while orange and white nodes correspond to places inside the standard library (respectively, `libc/libstdc++` and Linux' linker `ld-linux-x86-64`). Sink nodes correspond to the PC which resulted in a fault (used when grouping by PC). their out-degree. An edge from node $A \rightarrow B$ means that PC A called into a function, which eventually arrived at point B (whether because PC B caused a page fault, or B in turn called another function). Both nodes and edges use are sized logarithmically with respect to their frequency of appearance (i.e.: the bigger the node/thicker the edge, the exponentially more faults pass through it).
We create these graph visualizations using Cytoscape [53] and the yFiles [54] hierarchic layout for automatic node positioning (which we then manually adapt on some graphs to ensure node labels, i.e.: PCs, do not overlap). We include all the graphs (in `.gml` format) and our style file in our data repository. We recommend loading the graphs in Cytoscape for interactive

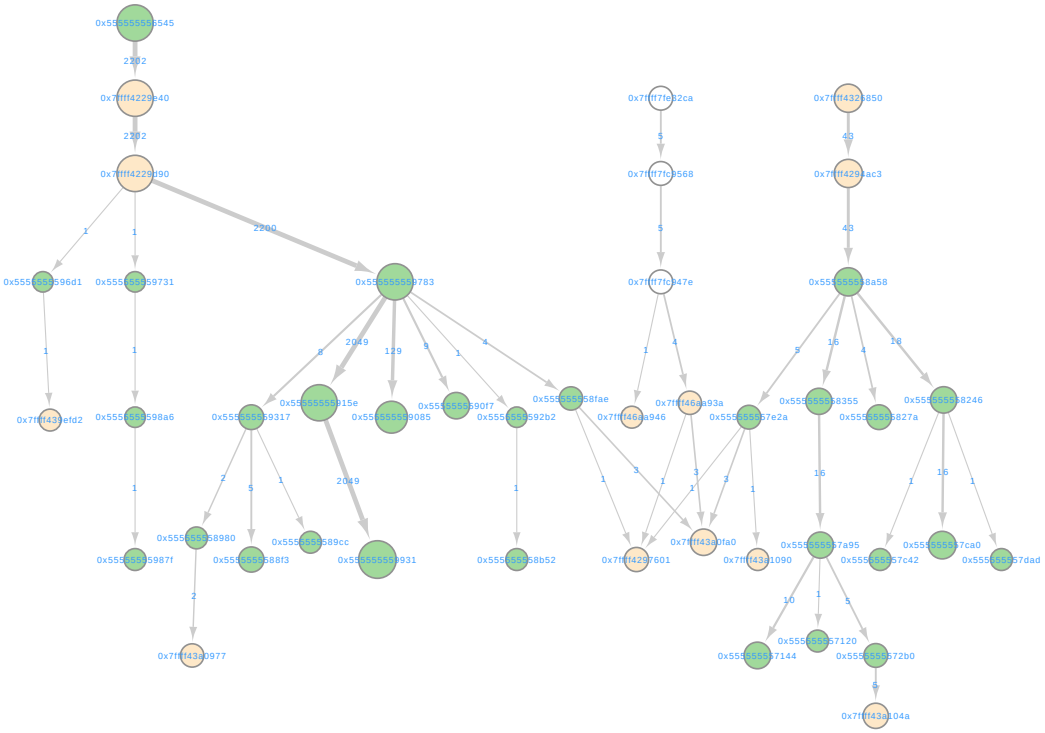


Figure 5: Example of an execution graph. Streamcluster, 25%

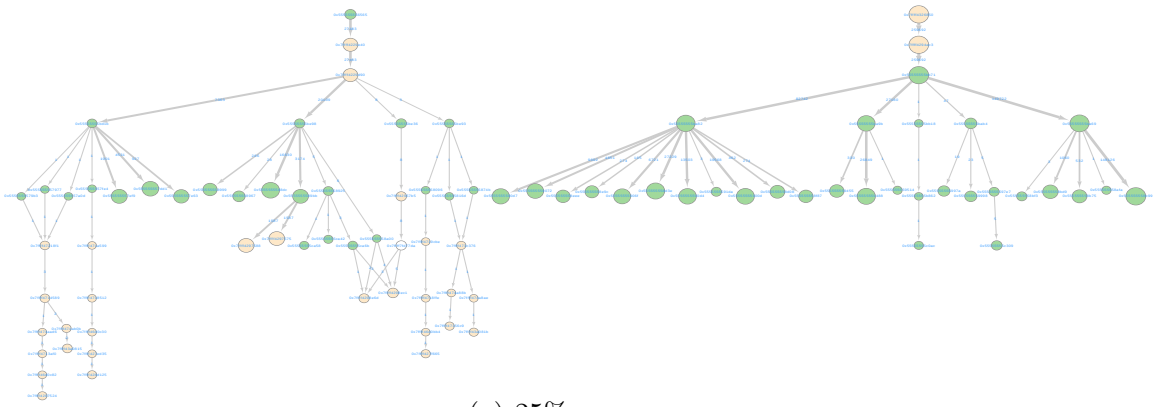
visualization, though we include an exhaustive copy of each of them in Appendix A for the sake of completeness.

Visualizing our data this way has two advantages. First, assuming access patterns can be mapped back to stacktraces as previously discussed, it gives a concrete representation of what this means, as one taken path in our graph (source to sink) corresponds to one such stacktrace group in our data. If we then manage to assign each path to one type of access pattern, or at least one type of behavior, this should make the task of prefetching easier. Secondly, this visualization of the traces enables us to compare different traces amongst each other. In the following examples, we will mainly focus on fluidanimate, as its graph size is comparatively smaller (in number of nodes), compared to the graphs of other benchmarks.

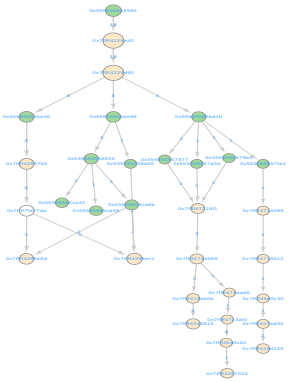
First, we compare multiple runs of the same application, but under different memory-pressure scenarios. For this comparison, we remove the cold misses, i.e. the first fault to a page. This results in smaller graphs (in number of nodes), but helps us focus on application behavior and abstract away application setup.

Figure 6 shows that, as one would expect, by increasing the memory pressure, we discover more parts of some graph subcomponents. This can especially be seen in the subcomponent on the left of the 25% scenario, whose size increases in terms of number of unique paths present. Then, we notice that some components appear as we increase the memory pressure, making a strong point towards a careful choice of experimental setup. Finally, we notice "important paths" (thicker edges) might change across memory pressure scenarios. In the 50% scenario, the edges from `0x7ffff4229d90` to `0x55555555bd1b` and to `0x55555555be98` appear as equally likely as the one to `0x55555555be36`, yet the former two see a substantial increase in size (i.e. the paths become more frequent) when going to the 25% scenario, while the other remains the same.

We continue by comparing two traces of the same application (fluidanimate) at the same memory pressure level (25% available memory) for two distinct runs of the application. The resulting graphs are shown in Figure 7.

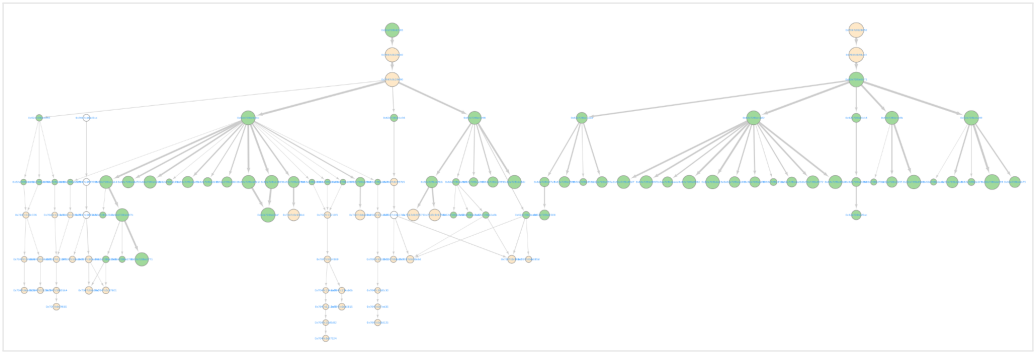


(a) 25%



(b) 50%

Figure 6: Comparison of fluidanimate, run at two different memory pressure levels



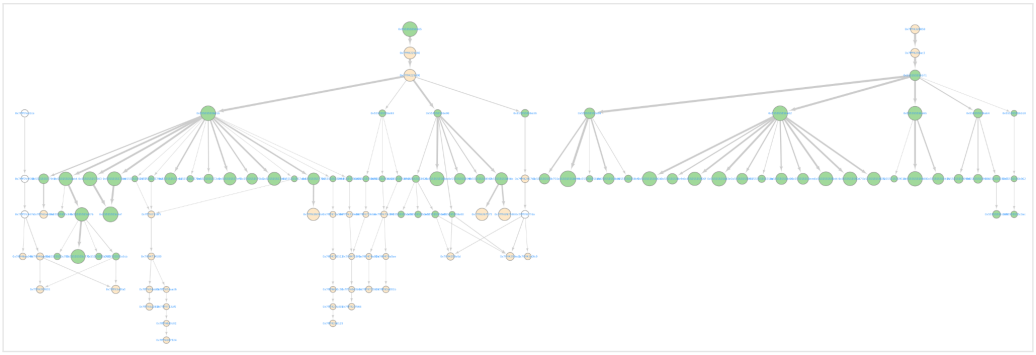
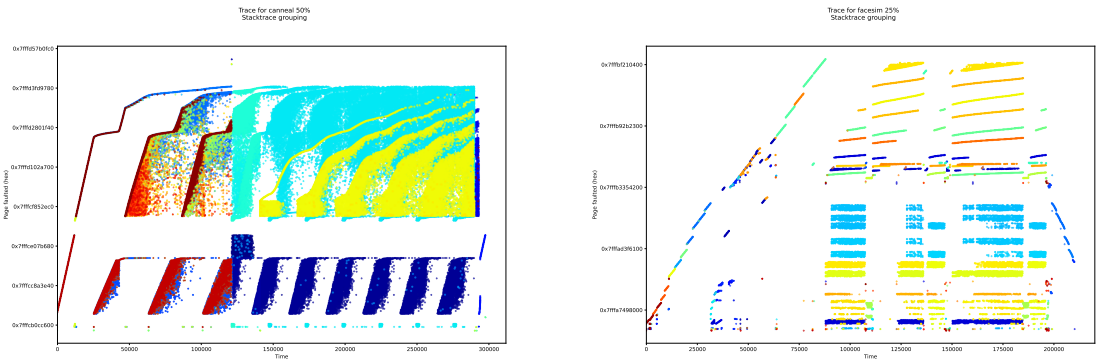


Figure 7: Two different runs of fluidanimate at the same memory pressure.

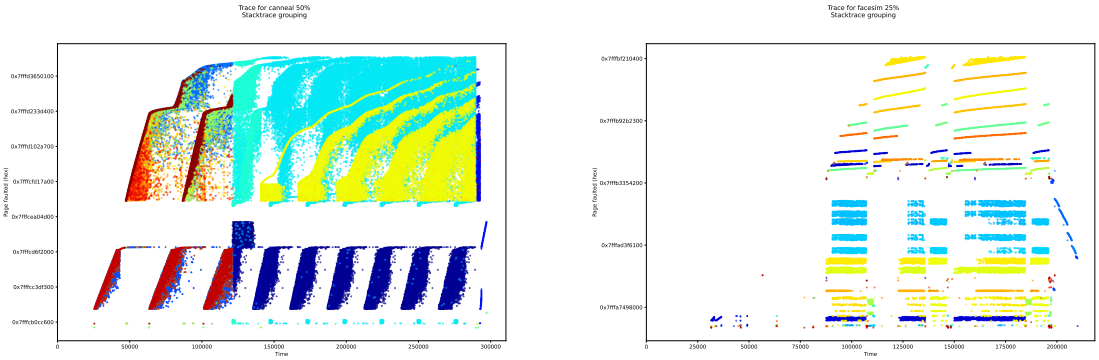
We can see that the graphs are very similar - the edge weights are overall the same and shape is similar. We thus deduce that, in our trace-gathering context, multiple runs of the same application at the same simulated memory pressure, will capture the same application semantics. As we will see in Section 4.2.2, this is important as it gives us a way to increase the dimensionality of our datasets if need be.

3.2.2 Grouped page fault traces

While there are advantages to execution graphs in modeling and understanding our data, because they abstract away the time dimension, they must only be used in complement to page fault traces. We can however also easily translate the *grouping by stacktrace* method to our time traces by simply coloring each page fault a different colour, based on the stacktrace, i.e. execution graph path it took to lead to the fault. Using this method, we readapt the traces from Figure 3 in Figure 8a. Similarly as for execution graphs, we include a list of all the grouped traces in Appendix B.



(a) Full memory trace



(b) Memory trace with the cold misses removed

Figure 8: Grouped page fault traces for canneal 50% (left) and facesim 25% (right)

Using this visualization of our traces, we can also better understand the impact of cold misses. As shown in Figure 8b, we can see that, without surprise, they are at the beginning of our traces. However, unlike what seemed to be an obvious decision from section 3.2.1 to remove them when comparing different traces, we see that the initial access pattern is sometimes also reflected in the future. For instance in canneal, we see the brown reverse-L shaped access stream repeat twice after the cold-miss period. This is interesting because cold misses are already handled differently inside the kernel, and it could thus lead to implementations of simpler systems for heuristic prefetching.

4 Revisiting Prefetching

4.1 Problem formulation

Traditionally, the problem of prefetching a (list) of addresses, when addr_t from address-space $\mathcal{A}$ faults at time t, can be modeled using formulation 1:

$$f(\text{addr}_{t-H}, \ldots, \text{addr}_{t-1}, \text{addr}_t) = \text{pred}_1, \text{pred}_2, \ldots, \text{pred}_K \Leftrightarrow f(\mathbf{y}_t) = \mathbf{y}_t^* \tag{1}$$

Here f is our prefetcher, H is the history size (i.e.: the number of past accesses we inform our prefetcher of), K is the prediction window (i.e.: the number of future accesses we predict), $\mathbf{y}_t \in \mathcal{A}^H$ are therefore a window of past memory accesses and $\mathbf{y}_t^* \in \mathcal{A}^K$ a list of K future predictions. Naturally, we can see the problem with this simple formulation as it does not permit for any additional information/features to be used. Implicitly, by using PCs as index into history tables, cache prefetchers like [15] assume the following formulation 2, where they have one prefetcher per PC.

$$f_{\texttt{PC}_i}(\mathbf{y}_t) = \mathbf{y}_t^* \tag{2}$$

Formulation 3 is analogous to formulation 2 and states that we have one prefetcher per stacktrace group/path in the execution trace. Formulation 4 generically states that we have an unique prefetcher, that also takes as input the stacktrace at fault time t. It is hard to imagine a heuristic prefetcher doing this and we will naturally be using it to automatically learn the usefulness of the stacktrace through ML.

$$f_{\texttt{stacktrace}_i}(\mathbf{y}_t) = \mathbf{y}_t^* \tag{3}$$

$$f(\mathbf{y}_t, \texttt{stacktrace}_t) = \mathbf{y}_t^* \tag{4}$$

Our work will consider all formulations 1 to 4. Note that, as stated in section 2.1.3 and identified by [16; 17], it is unclear whether formulation 2 (and analogously formulation 3) conceptually makes sense. Indeed, suppose you have the interleaved access stream described in Figure 9. Each vertical line on the time axis corresponds to one fault/miss. We colour the miss based on the group (PC, but the same argument holds if we group by stacktrace) this miss is linked to. The missed/faulted address is written above the vertical line. The red group performs a strided access pattern of stride +1, while the green group – of stride +2.

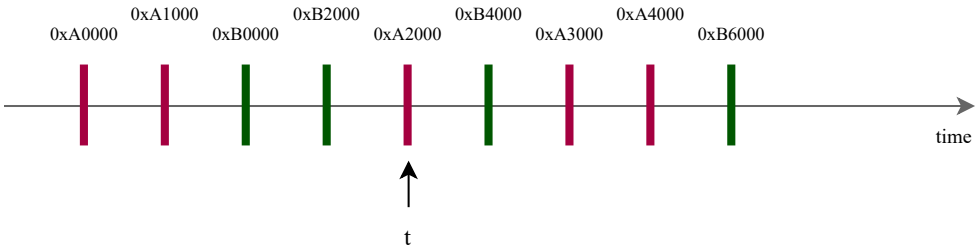


Figure 9: Toy example fault trace showcasing the problem of formulations 2 and 3

You are the prefetcher and have just observed the faulty address `0xA2000` at time t as pointed by the figure. Do you prefetch :

a. 0xA3000, because you are in the red group and realize that it has a stride of +1

b. 0xB4000, because you see that the most recent previous accesses were from the green group and therefore decide to follow its stride

c. both, as both red and green groups were relatively recent

d. something else/nothing, because you have computed deltas without performing any grouping (i.e. are colourblind with respect to the groups and only see gray bars), thus obtaining $\{1, 15, 2, -16\}$

In the above example, Leap would behave like item d.. We hypothesize that an ideal heuristic prefetcher would achieve best results if it were able to somehow separate the two streams. Conceptually, this would mean there is one prefetcher per group, and that prefetcher only acts on this group's accesses/misses. In turn, this would imply a sort of "timeline duplication" as each prefetcher is unaware of the others. We illustrate this in Figure 10.

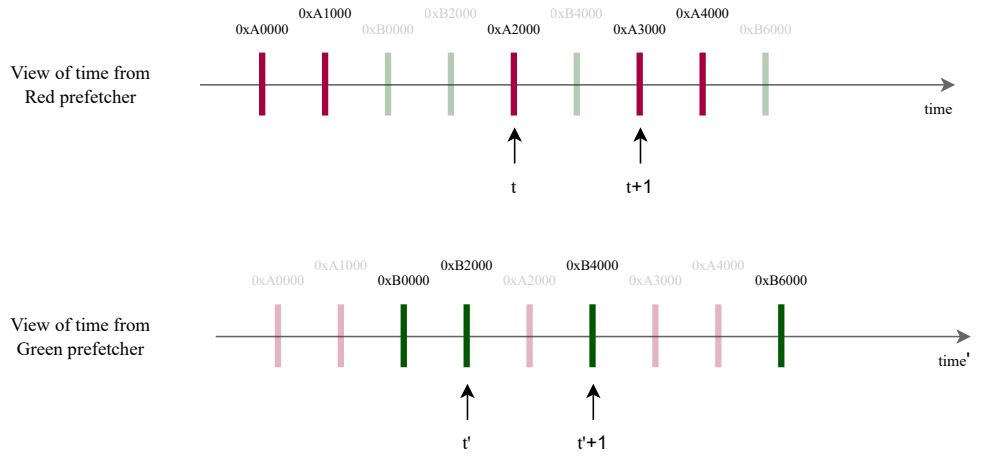


Figure 10: The "timeline duplication" – a hypothesized perfect scenario for using multiple prefetchers. Here, the top prefetcher is blind with respect to the greyed out, green marks which are only visible to the bottom prefetcher (and vice-versa).

While this seems like a good idea, it also shifts the problem – if you are at (ground truth) time t (as pictured in fig. 9), how do you you know that the next group will be green? Is this task easier than predicting the next address directly? To our knowledge this new task (e.g.: predicting the next faulty PC) has not been studied. In section 4.3, we will look at how a heuristic prefetcher like Leap, and a ML prefetcher like NLinear, can perform if put in the best conditions for this task, i.e.: assuming the "timeline duplication" shown in fig. 10. Then, in section 4.4 we will try to give an element of answer to the previous questions experimentally, by applying NLinear to it.

4.2 Evaluation setup and metrics

4.2.1 Leap modification

We use a slightly modified version of Leap where, instead of prefetching only when we are *certain enough* of our predicted Δ_{maj}, we always prefetch. That is, at each faulting address $\texttt{addr}_t$, we will predict some next K addresses as prefetch candidates. However, we log Leap's certainty at every prediction, formally defined as:

$$C_t = \frac{\sum_{i=1}^{H} [\![\Delta_{i,t} = \Delta_{\text{maj},t}]\!]}{H}$$

and where

$$[\![X]\!] = \begin{cases} 1, & \text{if X is true} \\ 0, & \text{otherwise} \end{cases}$$

denotes the Iverson bracket.

4.2.2 NLinear

To motivate the use of NLinear, let us quickly compare the data used in the field of LSTF and in our task.

Name	25%-trace size	Dimensions
bodytrack	10471	1
canneal	11038259	1
dedup	56162	1
facesim	210527	1
ferret	1350388	1
fluidanimate	366232	1
raytrace	63774	1
streamcluster	2252	1

(a) Benchmark dataset

Name	Dataset size	Dimensions
ECL	26211	321
ETTh	14307	7
ETTm	57507	7
Exchange	7207	8
Weather	52603	21
Traffic	17451	862

(b) LSTF datasets

Table 4: Datasets comparison between the LSTF and Prefetching domains

LSTF uses 5 typical datasets : electricity consumption (ECL) and oil temperature on a 15 minute, and 1h granularity (ETT{h,m}), Traffic flow, Weather forecasting, and stock Exchange (see [41] for example). Table 4 compares the sizes of the different datasets. In LSTF, *dimensions* refers to the number of time-series of the same data considered at the same time. For example, for ECL, one dimension is one client's electricity consumption, and the dataset therefore contains data for 321 clients. In our case, increasing the number of dimensions would be as trivial as tracing a benchmark many times, and stacking the results (on the "z" axis, in practice done by adding one dimension to the data). We can also see that our datasets are typically longer or, equivalently, more granular than those used for LSTF evaluation. An indicator for granularity-agnostic models can be seen in the comparison between ETTh and ETTm models. If a model performs better on ETTh for example, it might be a hint that sub-sampling our dataset could yield performance improvements. As NLinear achieves similar results on both datasets, we do no consider sub-sampling our data here.

While there is no dataset with a behavior as volatile as the one of our page fault traces described in section 3.2, the first 4 LSTF datasets exhibit higher seasonality than volatility (e.g: every morning at 08:00 there is more traffic, electricity usage is higher every night, it is cold every winter) which is less present in our data. We therefore consider the Stock exchange dataset as closest to our domain, and select a model which best performs on it. This is notably the case for NLinear.

4.2.3 Metrics

Typical cache prefetchers use two main metrics, namely

$$\text{Accuracy} = \frac{\text{Prefetched misses}}{\text{Total number of misses}}$$

and

$$\text{Coverage} = \frac{\text{Prefetched misses}}{\text{Total number of prefetches}}$$

Taken together, these two metrics capture well the cost of operation of a prefetcher as fetching-in new data into cache will come at the cost of having to remove some of the resident data, thus potentially leading to more faults in the future. To compute these metrics, we would thus need to either run our prefetcher on a live system, or simulate one. We are unable to do this with traces only, as discussed in section 5.

We therefore propose two new metrics, taking inspiration from the field of Document Retrieval, namely Precision at K ($P@K$) and Success.
Consider a page fault at time t. We denote the "ground truth" next T page faults used for evaluation by $\mathcal{T}_t = [\hat{y}_{t+1}, \hat{y}_{t+2}, \ \dots \ , \hat{y}_{t+T}]$. Then,

$$P@K_t = \frac{|\mathbf{y}_t^* \cap \mathcal{T}_t|}{K}$$

, where T=K

$$\text{Success}_t = [\![|\mathbf{y}_t^* \cap \mathcal{T}_t| \neq \emptyset \ ]\!]$$

, where T=1.
Intuitively, $P@K$ counts the number of future pages we manage to predict correctly, whereas Success is a binary variable saying whether or not we were able to predict the next page only. Both metrics are better when higher. In the next section, we report the mean of the metric evaluated at each iteration over the whole trace. When considering formulations 2 and 3, since we will get multiple such means, one per prefetcher, we report the weighted average of all those metrics, weighted by the group size the prefetcher was assigned to (i.e.: prefetchers prefetching in groups with more elements will impact the metric more).

4.3 Experiments description and results

For each benchmark, we consider their 25% trace. We perform three experiments for a prefetcher f. In the first one, the prefetcher on the whole trace. This serves as baseline for comparison. In the second one, we run one prefetcher per PC group (sink nodes of our execution graphs). Finally, in the third one, we run one per stacktrace group. In section 4.3.1, Leap is the prefetcher f and we perform the experiments on all the benchmarks. In section 4.3.2, NLinear is the prefetcher and we perform the experiments on only facesim, due to lack of time.

4.3.1 Leap

In this section, our prefetcher f is Leap. We set $H = K = 10$ Our aggregated results for the three experiments are shown in Figure 11.

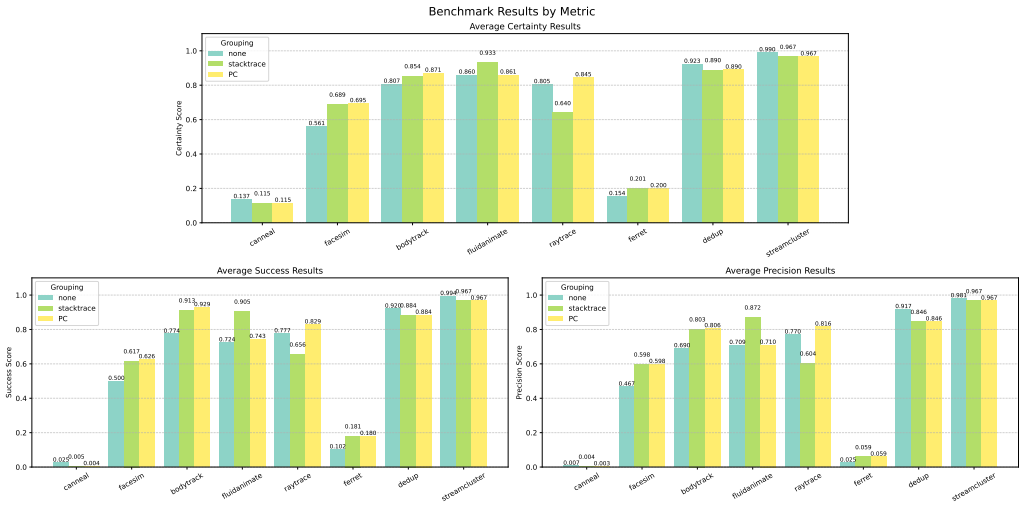


Figure 11: Results of our first experiment. "None" grouping stands for running Leap standalone on the full trace, whereas the other two grouping assume one Leap prefetcher per group.

Several elements stand out. First, surprisingly, despite the ideal scenario presented, the Leap prefetcher outperforms the group prefetcher in the cases of `canneal`,`dedup`,`streamcluster`. We hypothesize that the last two are caused by the issue that the traces were not too small for these benchmarks, as described in section 3.1. For `canneal`, this could mean that our understanding regarding the interleaving issue (fig. 9) might have been incorrect. In the other cases, we indeed see a performance improvement from our grouping approach. Secondly, we notice that the stacktrace grouping outperforms the PC grouping only in the case of `fluidanimate`, and achieves lower success and comparable precision otherwise. In the case of `raytrace`, it performs significantly worse. This suggests that our motivational toy example in fig. 4b might not be accurate representations of all applications.
To better understand what is going, we try to look into the groups themselves. Figure 12 reports, per group, the difference of the group's prefetcher with respect to the one from fig. 11. A positive (resp. negative) difference means that prefetcher for that group performed better (resp. worse) than the weighted average of all prefetchers.

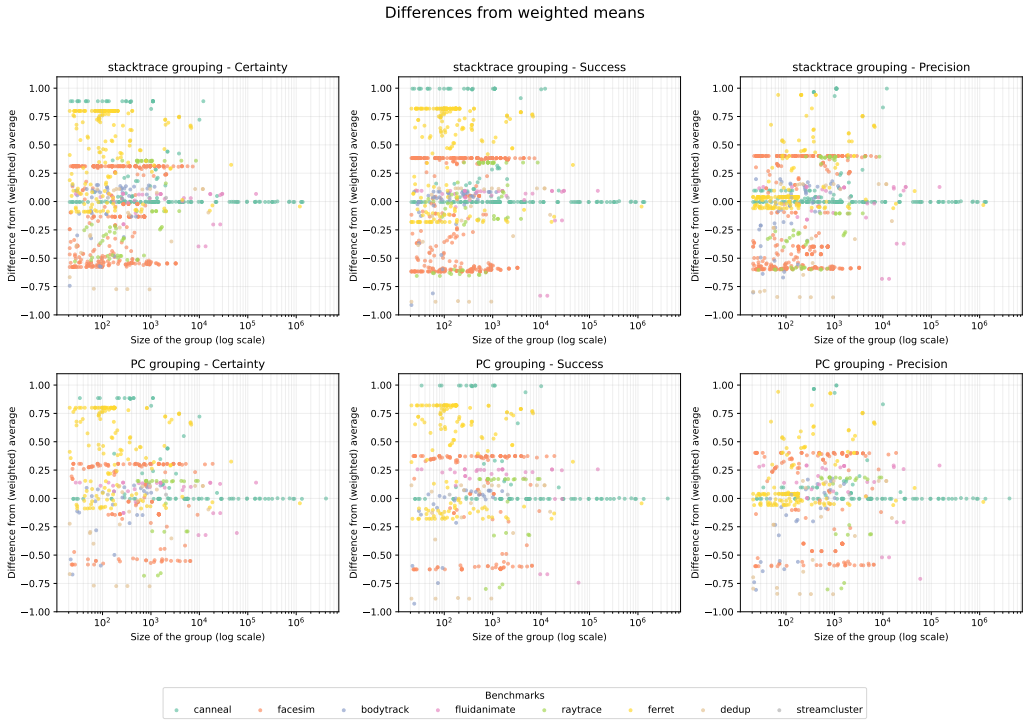


Figure 12: Difference from the weighted average metric reported in fig. 11. Each data point corresponds to one group, and is colored by the benchmark it comes from. Top line corresponds to stacktrace grouping, bottom line to the PC grouping

We notice that, for canneal, there are 30 (resp. 16) of stacktrace (resp. PC) groups which appear to perform very well comparatively. Upon closer inspection, these groups are of fairly small size however, ranging from 300 to 12000 elements, compared to the full trace size of 11M. Finally, as our choice of H is intuitively one of the defining factors for the prefetcher's performance, we also perform the same experiment with various choices for it. We keep $K = 10$. Figure 13 shows these results.
We notice that all prefetchers relatively quickly stabilize to the same results. While we cannot infer that the value of H is not important from this result only, it provides a good starting comparison.

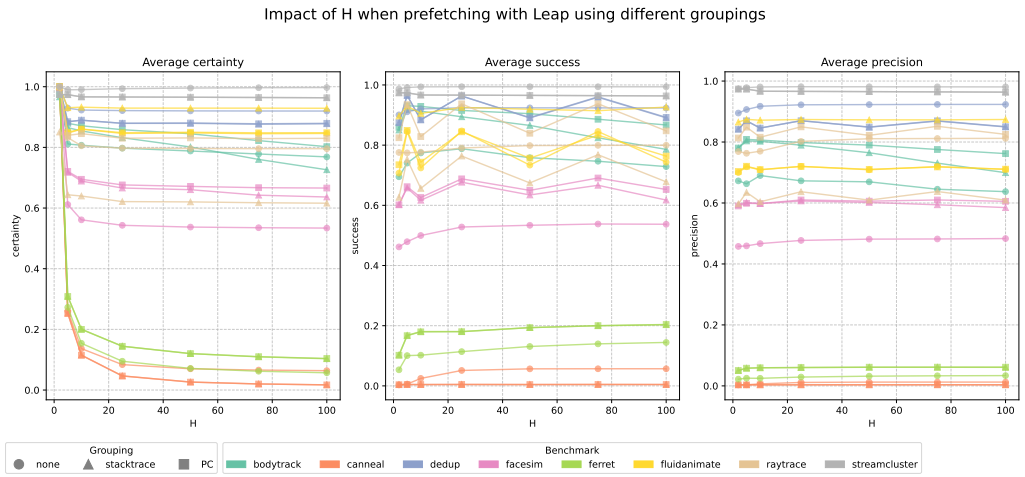


Figure 13: Results for our first experiment, for different value of H. Each benchmark is coloured in a different colour. Markers are used to specify whether the result was for grouped, or non-grouped runs, where a different marker is used to specify the group type.

4.3.2 NLinear

In this section, we run the three experiments using NLinear as a prefetcher. Due to lack of time, we only consider facesim 25%. Since we are now dealing with an ML model, there are several unknown hyperparameters we need to optimize for. We cross-validate different configrations and select the best performing one [5]. We directly compare its metrics results on the validation set to Leap's results from **??**.
We split our dataset in 75% for training and 25% for validation. As discussed in section 5 this methodology is flawed since the Leap result was obtained from running on the whole trace, whereas the validation set only corresponds to the last $\frac{1}{4}$ of the trace. It is nevertheless a good first point for comparison.
We propose results for two models, namely one with $H = 10$ for direct comparison to our Leap experiment, and one with a bigger history – $H = 96$. Figures 14 and 15 respectively show our results for success and precision.
Interesetingly, our model, despite being optimized for time-series, does not seem to perform as well as Leap at first sight. More thorough experiments will be required to confirm this however.

4.4 Insights on the feasibility of the new task

Finally, we also use the NLinear model to attempt predicting the next group, instead of the next faulty address. We use the same metrics as before. After cross-validation, the newly trained best model, with $H = K = 10$, is able at best to predict next PC with Success $= 0.4947$, $P@10 = 0.5018$, and stacktrace groups with Success $= 0.4624$, $P@10 = 0.4796$. This seems to point that the new way to model the task, detailed in , is of similar difficulty to the task of predicting addresses.

5 Limitations and Discussion

The biggest limitation of our work is described in section 3.1 – since the Linux readahead prefetcher marks (and fetches) predicted pages in the page cache, when they are accessed in

[5] The final configuration assumes $K = 10$, no data shuffling, deltas instead of raw addresses, a custom MultiStep LR Scheduler which linerarly decays from 10^{-2} to 10^{-4} through the epochs, 3 warmup epochs, 12 epochs in total, MSE as training loss

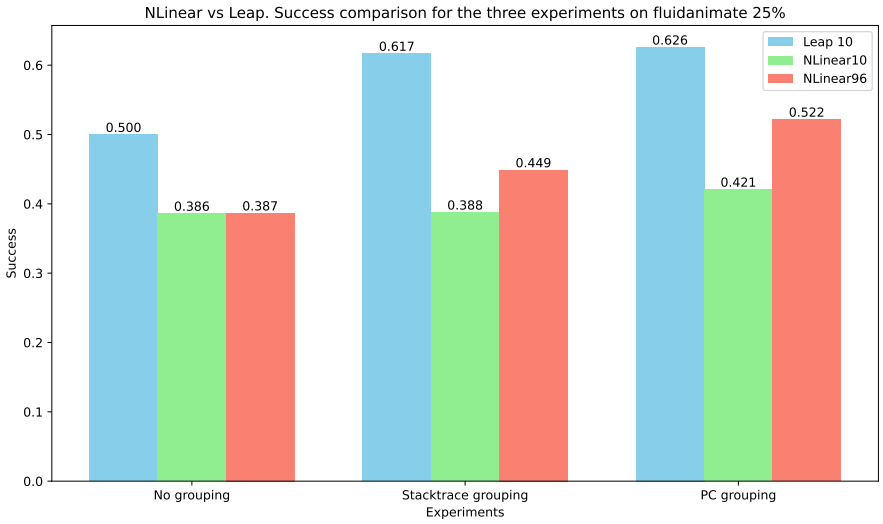


Figure 14: Success of NLinear with Leap for $K = 10$ predictions, $H = 10$ or $H = 96$ for NLinear on facesim 25%

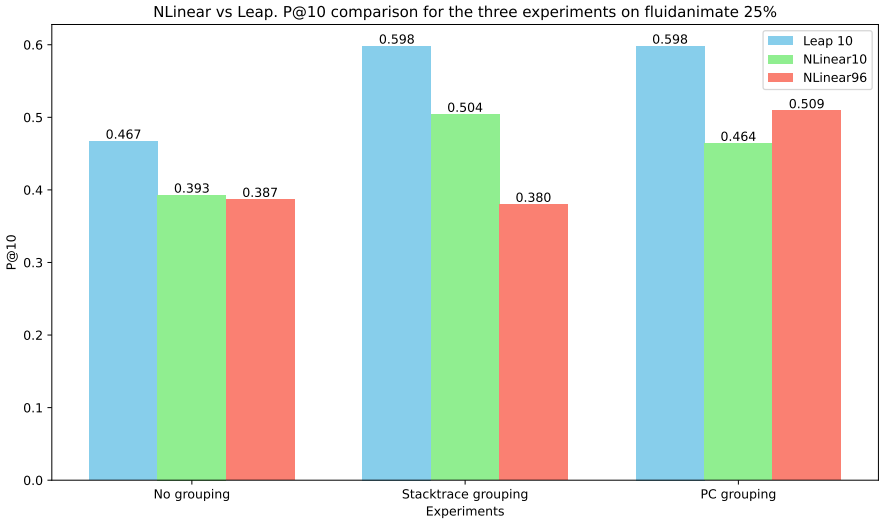


Figure 15: P@10 of NLinear with Leap for $K = 10$ predictions, $H = 10$ or $H = 96$ for NLinear on facesim 25%

the future, only a minor page fault happens. If it happens that `fltrace` does not capture minor page faults, this means that our traces are not traces of every page fault, as we would have liked, but rather of every page fault the Linux prefetcher did not manage to fetch. The impact of this is unclear - since the readahead mechanism is rather simple (fetches the closest next K pages), bigger strides and more complex patterns should still appear in our traces. It is however difficult to quantify to what extent this is true without performing tests manually. Two simple solutions exist to solve this problem:

1. Enable uffd MINOR_FAULT mode (though this requires the use of `shmem` or `hugetlbfs`. The former would increase the barrier of entry to gather new traces, as it would require running the tool on disaggregated memory servers. The latter would require adapting the rest of our work to huge pages.

2. Disable readahead on the machine doing trace gathering

The second option is more adequate and, in retrospect, should have been done to start with.

Secondly, we find the non-linear increase in trace size to be intriguing. Traces of benchmarks which theoretically were only allowed 50% to 75% of their used memry barely saw any page faults which weren't cold misses (i.e.: faults that also occur in the 100%). While this could be due to weird design choices in the benchmarks' code itself (e.g.: using big regions of memory in the beginning of the execution and never again afterwards, but never deallocating the arrays, thus making the tools we used to evaluate the application memory usage think that it is higher than what it actually is), it is also possible that the reported memory usage by `massif` and `/usr/bin/time` is inaccurate. Since there is a limit to how fine-grained our traces can be (e.g.: 10% traces make some of the benchmarks not complete execution), a "quick fix" would be to experiment gathering traces at other thresholds (e.g.: 30%,31%,32%,...) to see when a concrete increase starts, and how if evolves. Note that this is only feasible up to a point in practice, since memory allocations cannot naturally be of arbitrary size, `fltrace` makes the design choice of using 1MB as a unit (i.e. you cannot restrict memory to half megabytes).

When looking at execution graphs like the one in fig. 19a, we quickly see the limitations of the manual analysis done in section 3.2.1 as it is clearly unfeasible for bigger graphs. We must therefore find quantifiable ways to model graph changes. We have attempted computing graph edit distances, however, since this is an NP-hard problem, have abandoned the idea due to lack of time after letting the computation run for over 12 hours. A simpler idea would consist in looking at how many nodes overlap (by their label, program counter) and comparing their characteristics (e.g.: depth in the stacktrace, frequency relative to the nodes in-degree, ...) for the overlapping ones. Once again, due to lack of time, we could not explore this.

Furthermore, whether for the motivation or evaluation, our work is solely trace-driven. Previous work [6; 55] has done experimental studies (especially considering eviction) on RDMA and NVM systems' latencies which can be used to fill-in on some motivations. However, unlike for caches, there exists no easy solution for simulating complex systems which utilize all (or a mix of) the memory layers past Figure 1's (2). Despite our custom metrics, our trace-led evaluation is somewhat hard to interpret: to what extent are our predictions really useful, i.e. would prefetching what we predict lead to an application speedup? When we correctly predict a more distant page fault, will this successful prediction repeat in subsequent predictions, thus biasing our results since we would count it twice ? E.g., if we correctly predict $page_{t+4}$ at page fault time t, will we correctly predict $page_{t'+3}$ (which is equal to $page_{t+4}$ because our traces are *static* – we don't update them after prefetching) at time time $t' = t$? If yes, is this good, or

bad? One could easily think of system designs where a page is prefetched only if it has been predicted a thresholds amount of times before. While evidence to these questions would be to implement and test such prefetchers on live systems (e.g.: using FetchBPF (UBC) [56] for easier integration) , we cannot deny that trace-led development leads to faster research iterations, and is even required for implementing ML-based components of our systems. Traces however inherently give us only a snapshot of applications' runs. Even if one manages to develop complex heuristics or models to capture them, as we have seen in section 3.2.2, these patterns might disappear as memory pressure increases (or, for live systems, as the application is influenced by other components (other applications, response to user input,...) of the system) – would it thus still be interesting to implement such a system which would only work a fraction of the time? We leave this question unanswered.

In addition, we have gathered an extensive multi-feature dataset in response to difference 2.. However, we have had the time to apply extensive analysis only on one of the new features – the stacktrace. As such, we only use a subset of the features described by section 3.1. If we wanted to use more features for prefetching, we would also need a more generic problem formulation than those described in section 4. Formulation 5 would be an example of it.

$$f(\mathbf{y}_t, \texttt{stacktrace}_t, \texttt{registers}_t, \texttt{flags}_t, \texttt{code}) = \mathbf{y}_t^* \tag{5}$$

While we have started data analysis on other of the features in parallel, such as by looking at the registers values change over, grouping them by stacktrace as done in section 3.2 and then clustering them, we have not managed to find an interesting insight from this feature yet, and thus leave this out of the report. We leave this, as well as the task of analysing the potential impact of the *flags* (read, or write to page) and *surrounding code* features, and using them for prefetching, to future work.
Finally, while we have considered modeling the problem directly through existing machine learning, we hypothesize taking a step back into statistics can be insightful. Indeed, some key concepts from stochastic processes are applicable to our problem. For instance, martingales, which form the bases of financial probabilistic theories, Ornstein–Uhlenbeck, and Wiener processes could be great candidates for modeling our data. The applications which exhibit no autocorrelation, such as `canneal`, could on the other hand be modeled using Copula processes. Modeling data this way would require finding a set of data hyperparameters which fit the dynamics of the desired model, a different task which could then be done through ML or experimentally.

6 Conclusion

To conclude, our work was motivated by the complex task of building a memory-configuration-agnostic OS page-prefetcher. For this, we have gathered a multi-featured datasets composed of page-fault traces for several of the PARSEC benchmarks. Unfortunately, at this important building-block step for the rest of our work, we have failed to take into consideration Linux's `readahead` prefetching mechanism, thus rendering the rest of our work potentially meaningless. After gathering the traces, we continued by providing a novel way to visualize them, namely through execution graphs and grouped page fault traces. This let us visually identify some patterns in what appeared to be complicated traces at first. With the goal of predicting those patterns, we propose a new formulation for the prefetching task. Namely, we propose having one prefetcher per group (where a group can be either at the PC, or stacktrace granularity). We evaluate our approach using Leap as baseline prefetcher, and also consider NLinear, a time-series ML model composed of only one linear layer. When comparing our experiment

results, our proposed approach of stacktrace grouping performs better than Leap on the facesim, bodytrack, fluidanimate, and ferret benchmarks, with respect to both Success and Recall, two new metrics we introduce for trace-led evaluation. Noteably, it reaches a 25% success and 23% precision improvement over base Leap for the fluidanimate benchmark restricted at 25% memory. However, it reaches performance degradation with respect to the streamcluster, canneal, and dedup benchmarks. We then proceed to also evaluate NLinear on the same task and find it consistently performs worse than basic Leap for page prediction. We finally also test NLinear for the new task of "next group prediction" (i.e.: predict at page fault time t what the next PC, or path, yielding a page fault will be). It predict the next PC with Success = 0.4947, and $P@10 = 0.5018$, and predicts the stacktrace group with Success = 0.4624, and $P@10 = 0.4796$.

References

[1] H. A. Maruf and M. Chowdhury, "Effectively Prefetching Remote Memory with Leap," pp. 843–857, 2020.

[2] A. Zeng, M. Chen, L. Zhang, and Q. Xu, "Are Transformers Effective for Time Series Forecasting?," Aug. 2022. arXiv:2205.13504 [cs].

[3] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, pp. 20–24, Mar. 1995.

[4] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*, CF '04, (New York, NY, USA), p. 162, Association for Computing Machinery, Apr. 2004.

[5] C. A. Mack, "Fifty Years of Moore's Law," *IEEE Transactions on Semiconductor Manufacturing*, vol. 24, pp. 202–207, May 2011. Conference Name: IEEE Transactions on Semiconductor Manufacturing.

[6] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, (Virtual Event Germany), pp. 392–407, ACM, Oct. 2021.

[7] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos, "TMO: transparent memory offloading in datacenters," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, (New York, NY, USA), pp. 609–621, Association for Computing Machinery, 2022.

[8] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "TPP: Transparent Page Placement for CXL-Enabled Tiered Memory," June 2022. arXiv:2206.02878 [cs].

[9] D. Moura, V. Petrucci, and D. Mosse, "Performance Characterization of AutoNUMA Memory Tiering on Graph Analytics," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 171–184, Nov. 2022. arXiv:2212.04344 [cs].

[10] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?," in *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, (New York, NY, USA), pp. 1–16, Association for Computing Machinery, 2020.

[11] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient Memory Disaggregation with Infiniswap," pp. 649–667, 2017.

[12] J. Lee, H. Kim, and R. Vuduc, "When Prefetching Works, When It Doesn't, and Why," *ACM Trans. Archit. Code Optim.*, vol. 9, pp. 2:1–2:29, Mar. 2012.

[13] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, (New York, NY, USA), pp. 176–186, Association for Computing Machinery, Aug. 1991.

[14] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 469–480, Mar. 2016. ISSN: 2378-203X.

[15] S. Kondguli and M. Huang, "T2: A Highly Accurate and Energy Efficient Stride Prefetcher," in *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 373–376, Nov. 2017. ISSN: 1063-6404.

[16] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 141–152, Dec. 2015. ISSN: 2379-3155.

[17] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, Oct. 2016.

[18] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, (New York, NY, USA), pp. 252–263, Association for Computing Machinery, May 1997.

[19] T. Alexander and G. Kedem, "Distributed prefetch-buffer/cache design for high performance memory systems," in *Proceedings. Second International Symposium on High-Performance Computer Architecture*, pp. 254–263, Feb. 1996.

[20] K. Nesbit and J. Smith, "Data Cache Prefetching Using a Global History Buffer," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pp. 96–96, Feb. 2004. ISSN: 1530-0897.

[21] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 247–259, Association for Computing Machinery, Dec. 2013.

[22] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 7–17, Feb. 2009. ISSN: 2378-203X.

[23] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," *SIGPLAN Not.*, vol. 27, pp. 62–73, Sept. 1992.

[24] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying Memory Access Patterns for Prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), pp. 513–526, Association for Computing Machinery, Mar. 2020.

[25] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 178–190, Association for Computing Machinery, 2015.

[26] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial Memory Streaming," *SIGARCH Comput. Archit. News*, vol. 34, no. 2, pp. 252–263, 2006.

[27] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning Memory Access Patterns," Mar. 2018. arXiv:1803.02329 [cs, stat].

[28] A. Srivastava, A. Lazaris, B. Brooks, R. Kannan, and V. K. Prasanna, "Predicting memory accesses: the road to compact ML-driven prefetcher," in *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, (New York, NY, USA), pp. 461–470, Association for Computing Machinery, Sept. 2019.

[29] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, (New York, NY, USA), pp. 861–873, Association for Computing Machinery, Apr. 2021.

[30] M. Wu, K. Joshi, A. Sheinberg, G. Cox, A. Khandelwal, R. P. Pothukuchi, and A. Bhattacharjee, "Prefetching Using Principles of Hippocampal-Neocortical Interaction," in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, (Providence RI USA), pp. 53–60, ACM, June 2023.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," June 2017. arXiv:1706.03762 [cs].

[32] P. Zhang, A. Srivastava, A. V. Nori, R. Kannan, and V. K. Prasanna, "TransforMAP: Transformer for Memory Access Prediction," May 2022. arXiv:2205.14778 [cs].

[33] "readahead(2) - Linux manual page."

[34] S. Jiang, X. Ding, Y. Xu, and K. Davis, "A Prefetching Scheme Exploiting both Data Layout and Access History on Disk," *ACM Trans. Storage*, vol. 9, no. 3, pp. 10:1–10:23, 2013.

[35] B. Dong, X. Zhong, Q. Zheng, L. Jian, J. Liu, J. Qiu, and Y. Li, "Correlation Based File Prefetching Approach for Hadoop," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pp. 41–48, Nov. 2010.

[36] A. J. Uppal, R. C. Chiang, and H. H. Huang, "Flashy prefetching for high-performance flash drives," in *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, Apr. 2012. ISSN: 2160-1968.

[37] W. Fengguang, X. Hongsheng, and X. Chenfeng, "On the design of a new Linux readahead framework," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 75–84, 2008.

[38] A. Laga, J. Boukhobza, M. Koskas, and F. Singhoff, "Lynx: a learning linux prefetching mechanism for SSD performance model," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 1–6, Aug. 2016.

[39] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu, "Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory," Apr. 2023.

[40] H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, and W. Zhang, "Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting," Mar. 2021. arXiv:2012.07436 [cs].

[41] Y. Liu, T. Hu, H. Zhang, H. Wu, S. Wang, L. Ma, and M. Long, "iTransformer: Inverted Transformers Are Effective for Time Series Forecasting," Mar. 2024. arXiv:2310.06625 [cs].

[42] H. Wu, J. Xu, J. Wang, and M. Long, "Autoformer: decomposition transformers with auto-correlation for long-term series forecasting," in *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, (Red Hook, NY, USA), pp. 22419–22430, Curran Associates Inc., June 2024.

[43] T. Kim, J. Kim, Y. Tae, C. Park, J.-H. Choi, and J. Choo, "REVERSIBLE INSTANCE NORMALIZATION FOR ACCURATE TIME-SERIES FORECASTING AGAINST DISTRIBUTION SHIFT," 2022.

[44] T. Zhou, Z. Ma, Q. Wen, X. Wang, L. Sun, and R. Jin, "FEDformer: Frequency Enhanced Decomposed Transformer for Long-term Series Forecasting," in *Proceedings of the 39th International Conference on Machine Learning*, pp. 27268–27286, PMLR, June 2022. ISSN: 2640-3498.

[45] Intel, "CHAPTER 20 PERFORMANCE MONITORING," in *Intel® 64 and IA-32 Architectures Software Developer's Manual*, vol. 3B, pp. 3713–3868, Dec. 2022.

[46] "Perf Wiki," 2015.

[47] Intel, "Pin - A Dynamic Binary Instrumentation Tool."

[48] "SystemTap."

[49] "userfaultfd(2) - Linux manual page."

[50] A. Yelam, S. Grant, E. Liu, R. N. Mysore, M. K. Aguilera, A. Ousterhout, and A. C. Snoeren, "Limited Access: The Truth Behind Far Memory," WORDS '23, (New York, NY, USA), pp. 37–43, Association for Computing Machinery, 2023. event-place: Koblenz, Germany.

[51] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, (New York, NY, USA), pp. 72–81, Association for Computing Machinery, Oct. 2008.

[52] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, pp. 2:1–2:34, Mar. 2012.

[53] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker, "Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks," *Genome Research*, vol. 13, pp. 2498–2504, Nov. 2003.

[54] R. Wiese, M. Eiglsperger, and M. Kaufmann, "yFiles — Visualization and Automatic Layout of Graphs," in *Graph Drawing Software* (M. Jünger and P. Mutzel, eds.), pp. 173–191, Berlin, Heidelberg: Springer, 2004.

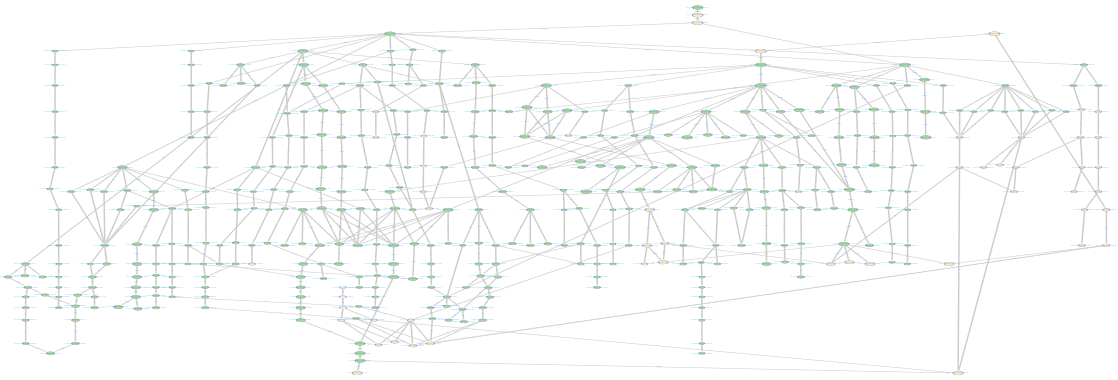
[55] P. Duraisamy, W. Xu, S. Hare, R. Rajwar, D. Culler, Z. Xu, J. Fan, C. Kennelly, B. McCloskey, D. Mijailovic, B. Morris, C. Mukherjee, J. Ren, G. Thelen, P. Turner, C. Villavieja, P. Ranganathan, and A. Vahdat, "Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale," in *Proceedings of the 28th ACM International Conference*

on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, (New York, NY, USA), pp. 727–741, Association for Computing Machinery, Mar. 2023.

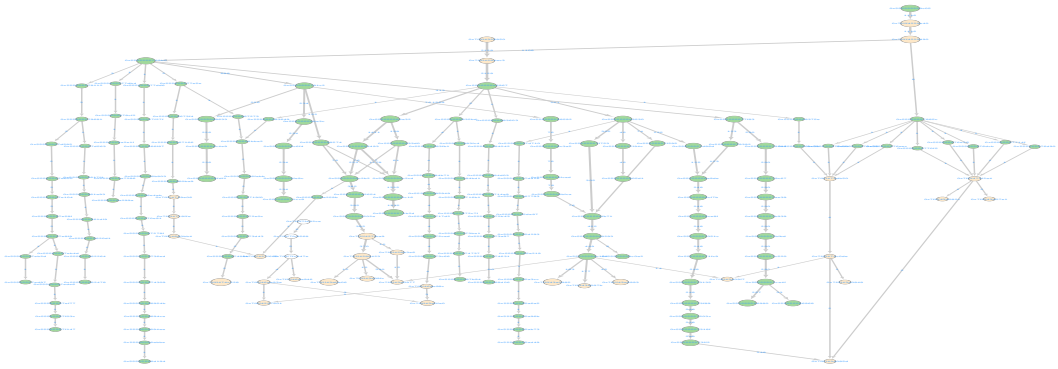
[56] X. Cao, S. Patel, S. Y. Lim, X. Han, and T. Pasquier, "{FetchBPF}: Customizable Prefetching Policies in Linux with {eBPF}," pp. 369–378, 2024.

A Execution graphs for PARSEC benchmarks

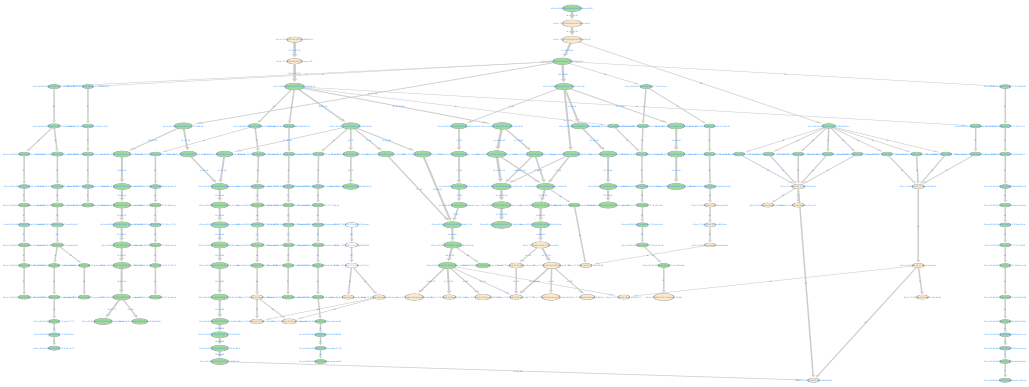
The following next few pages contains the execution graphs for all PARSEC benchmarks considered in our work, at different simulated levels of memory pressure. See sections 3.1 and 3.2.1 for more information. While we include these graphs here for the sake of completion, we recommend viewing them interactively through Cytoscape as described in section 3.2.1. Zooming in too much on the report will likely crash your PDF viewer.



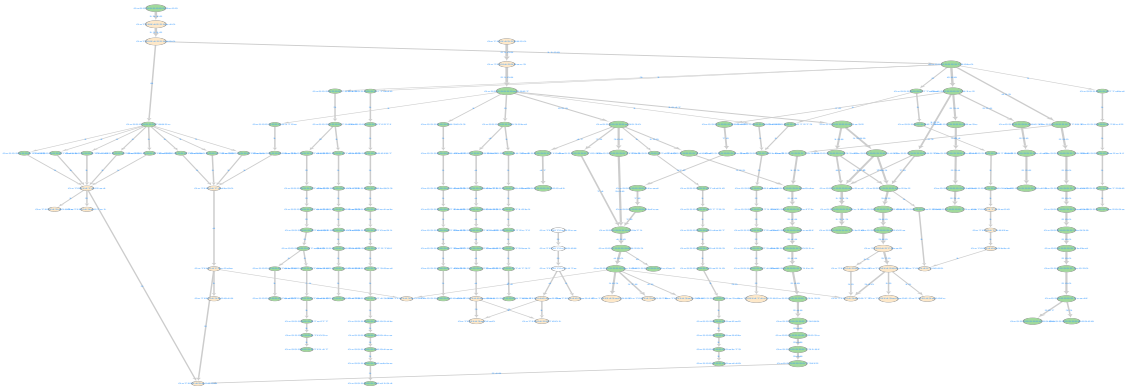
(a) 25%



(b) 50%

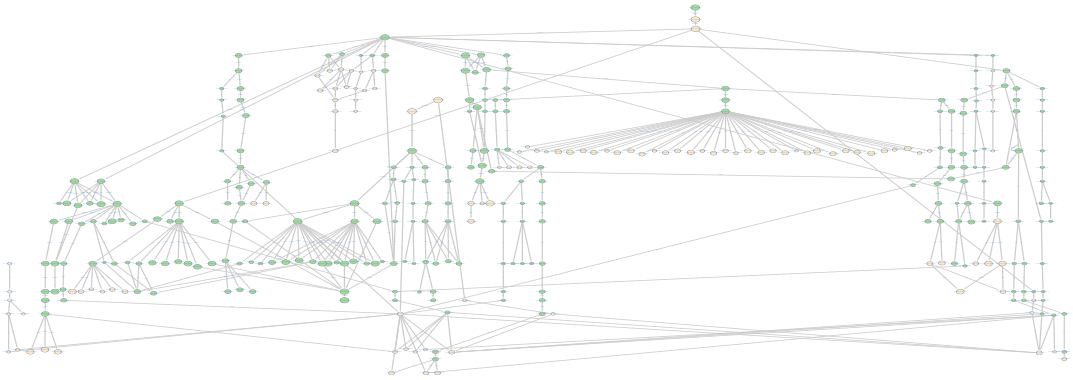


(c) 75%

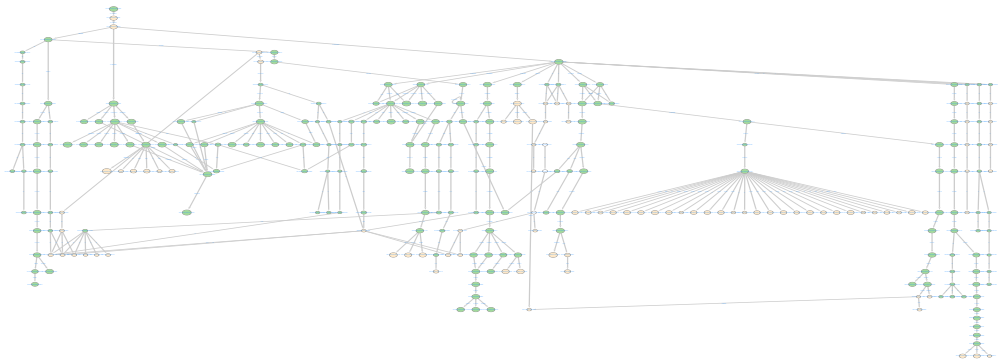


(d) 100%

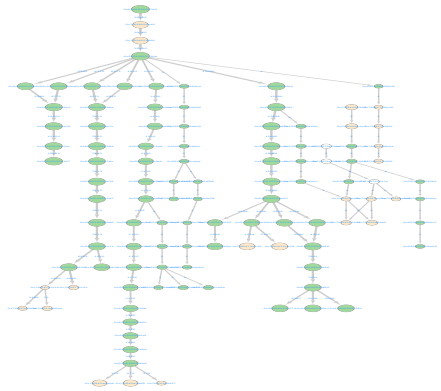
Figure 16: Bodytrack



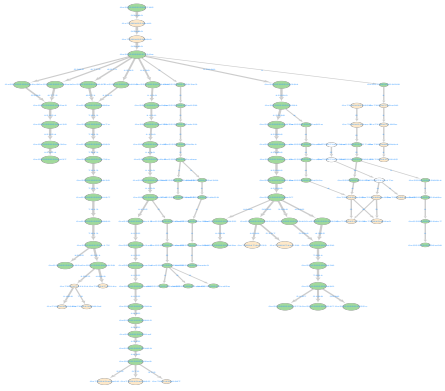
(a) 25%



(b) 50%

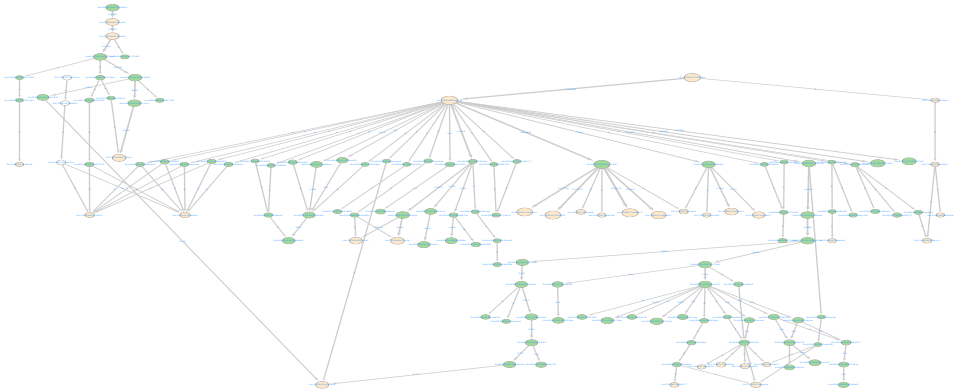


(c) 75%

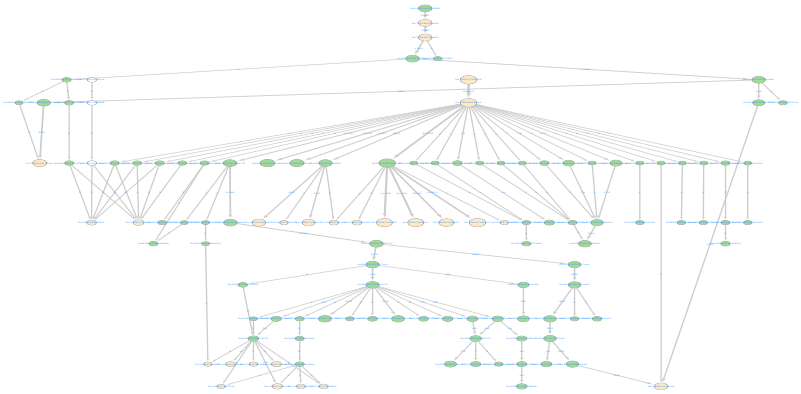


(d) 100%

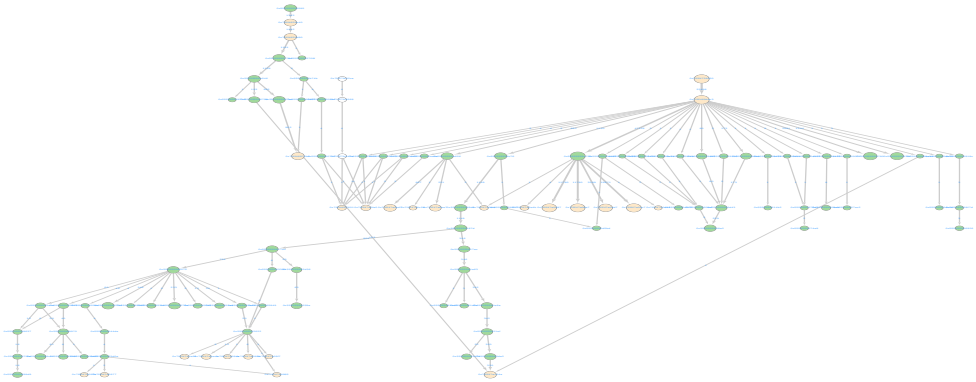
Figure 17: Canneal



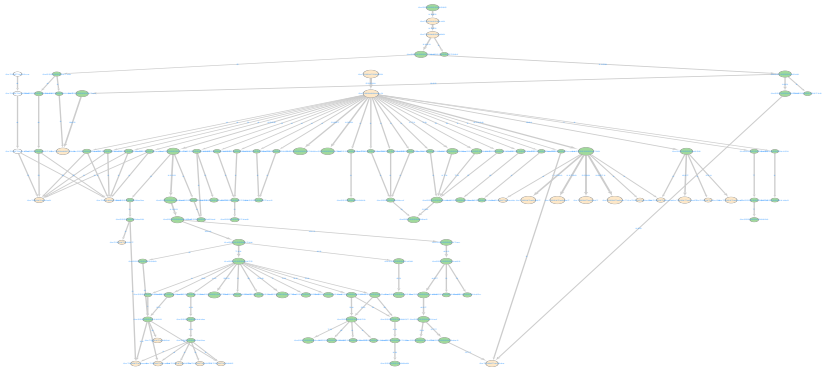
(a) 25%



(b) 50%

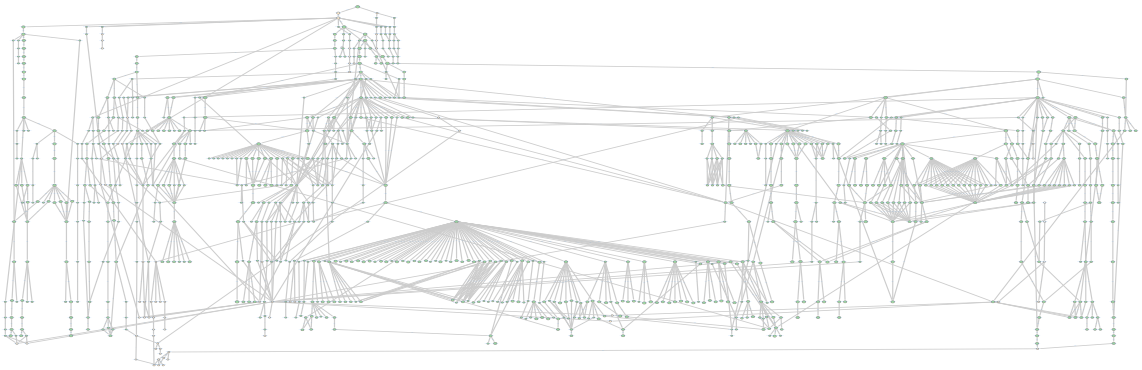


(c) 75%

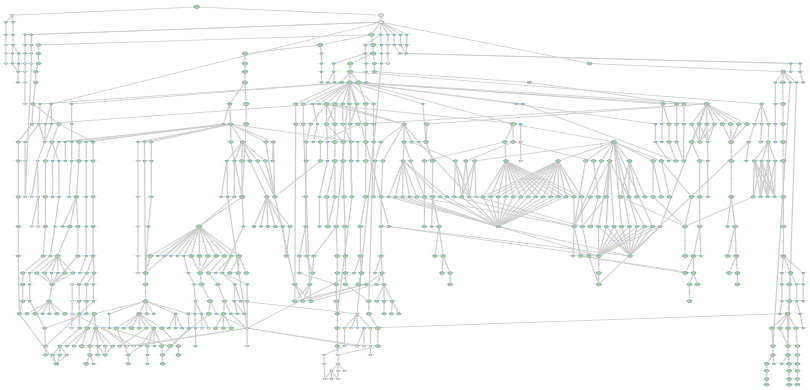


(d) 100%

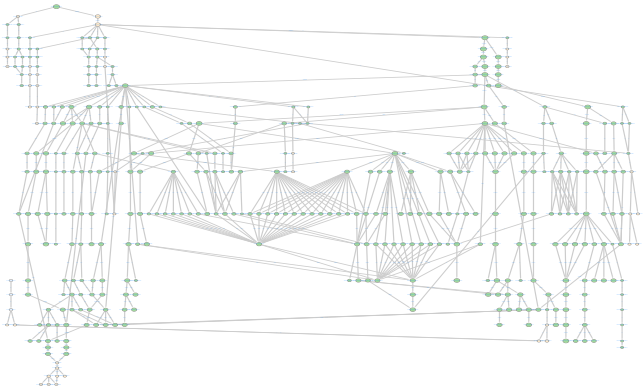
Figure 18: Dedup



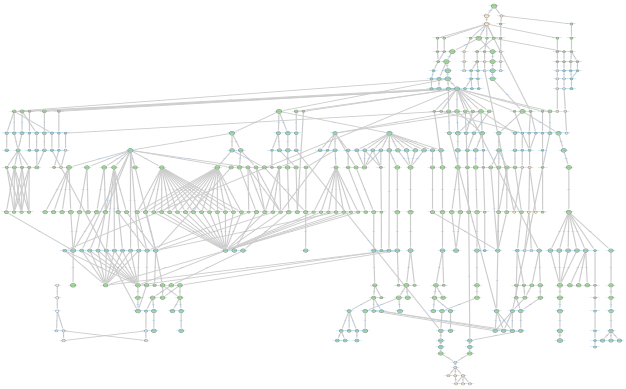
(a) 25%



(b) 50%

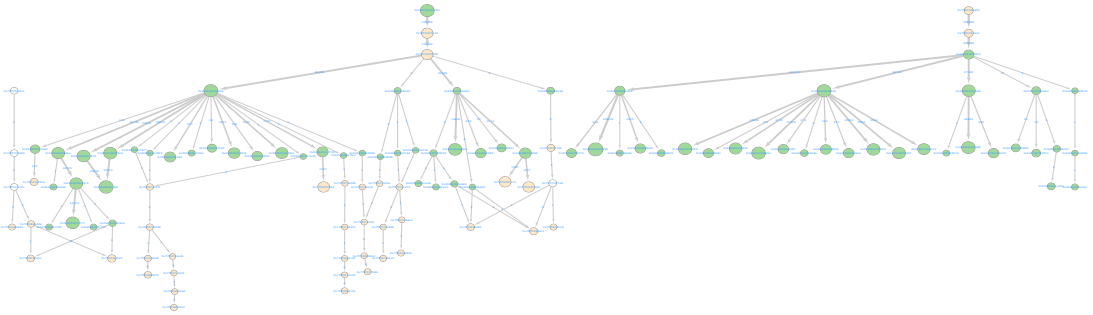


(c) 75%

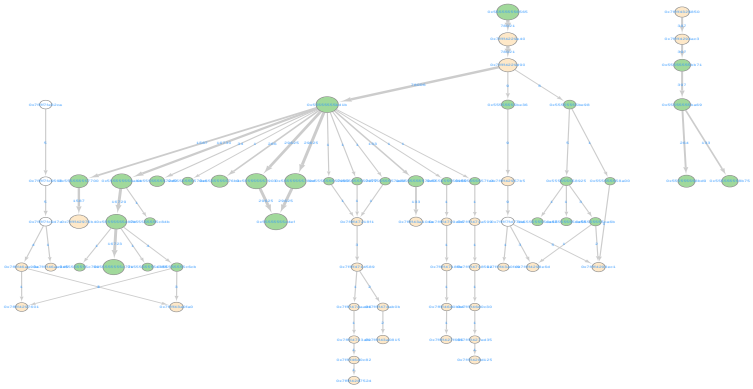


(d) 100%

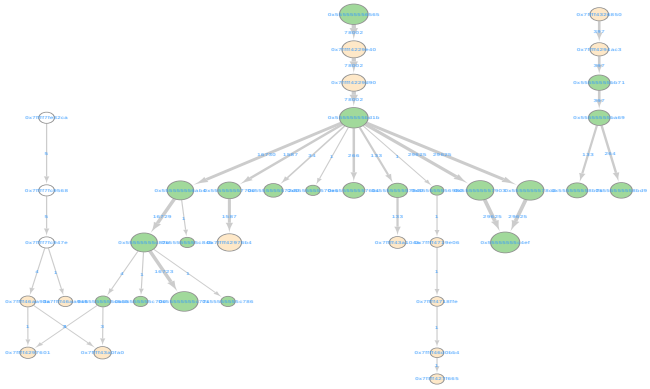
Figure 19: Facesim



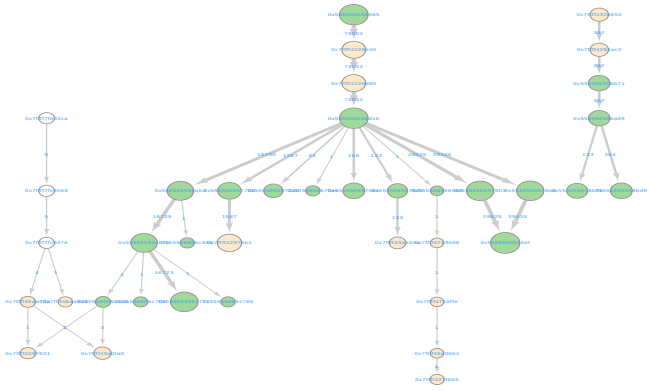
(a) 25%



(b) 50%

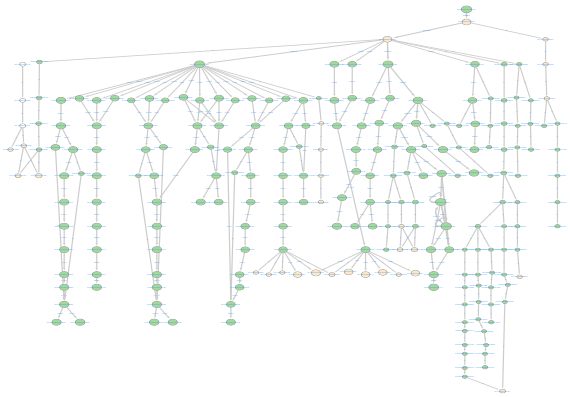


(c) 75%

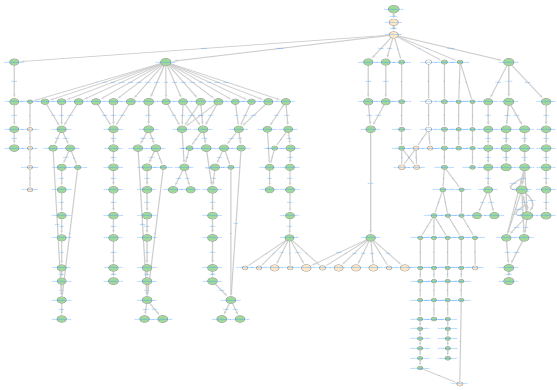


(d) 100%

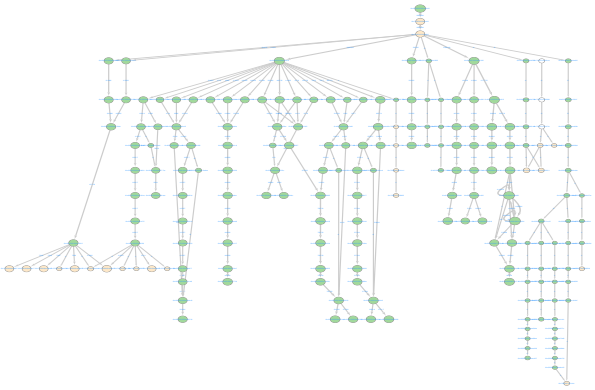
Figure 20: Fluidanimate



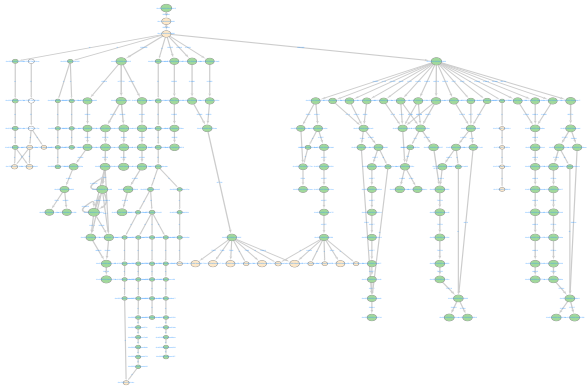
(a) 25%



(b) 50%

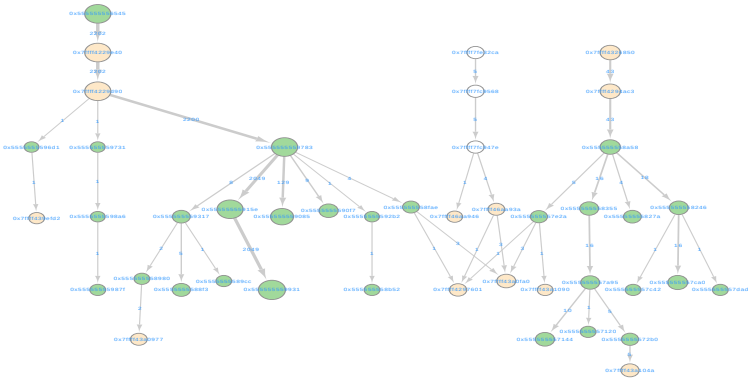


(c) 75%

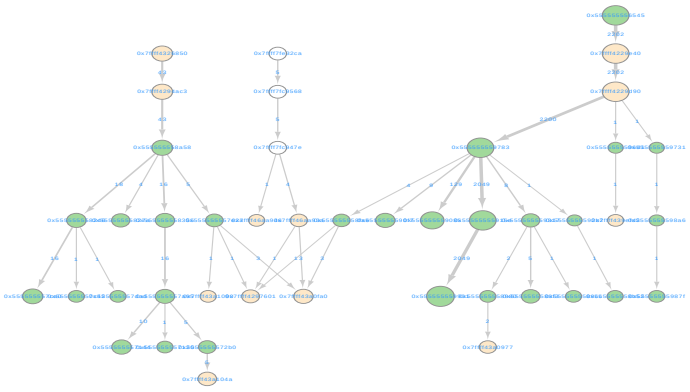


(d) 100%

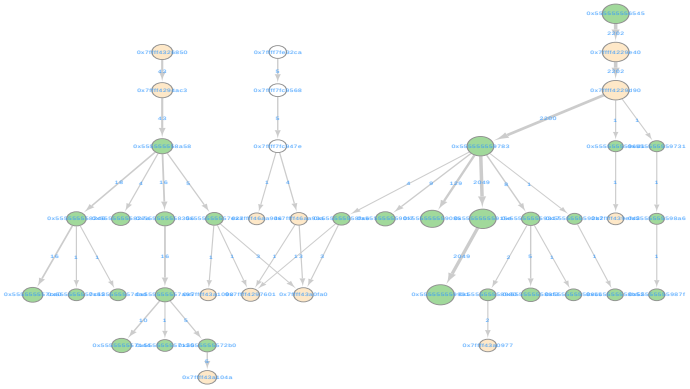
Figure 21: Raytrace



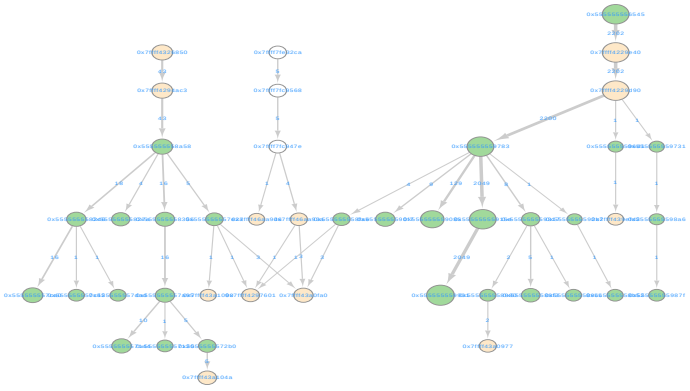
(a) 25%



(b) 50%



(c) 75%

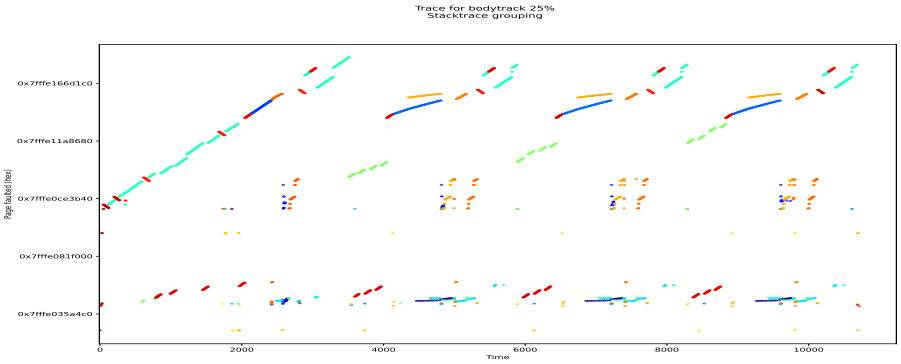


(d) 100%

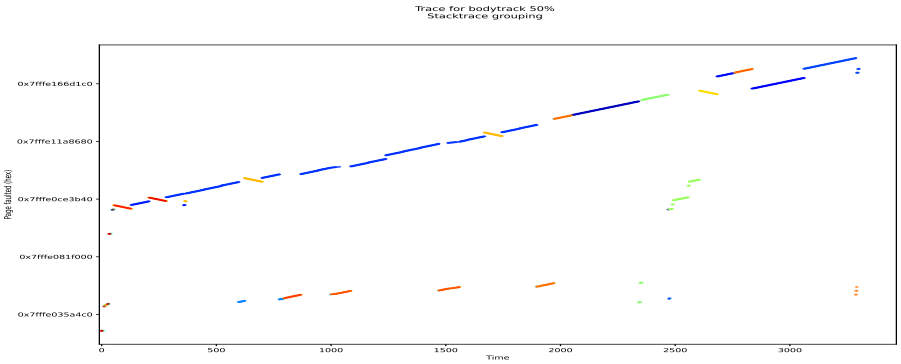
Figure 22: Streamcluster

B Stacktrace-grouped memory traces for PARSEC benchmarks

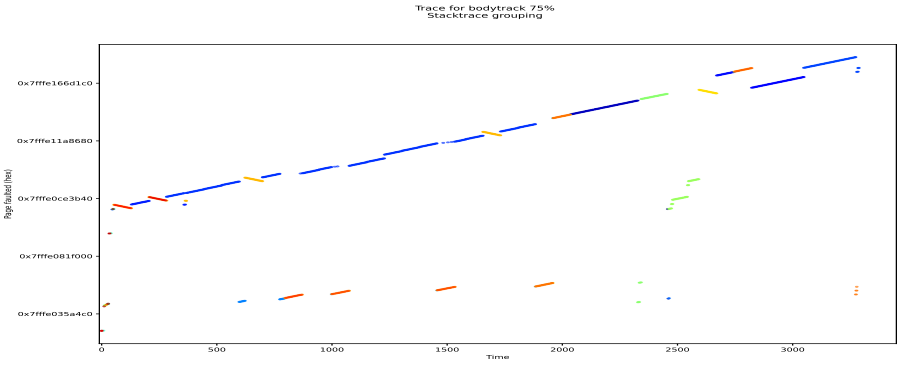
Similarly to Appendix A, we show in this section all the page fault traces, coloured as described in section 3.2.2.



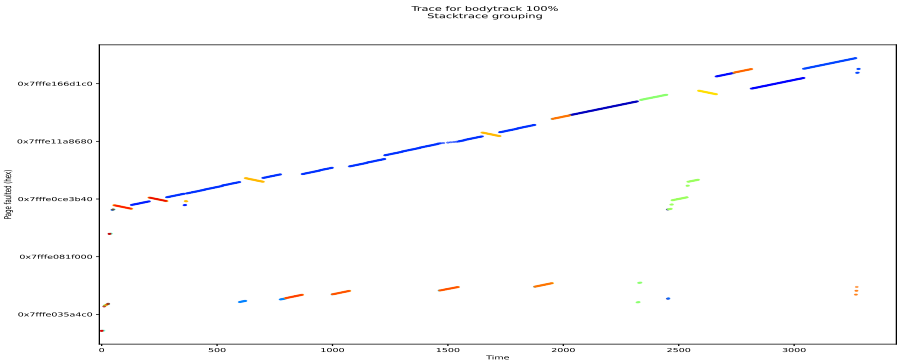
(a) 25%



(b) 50%

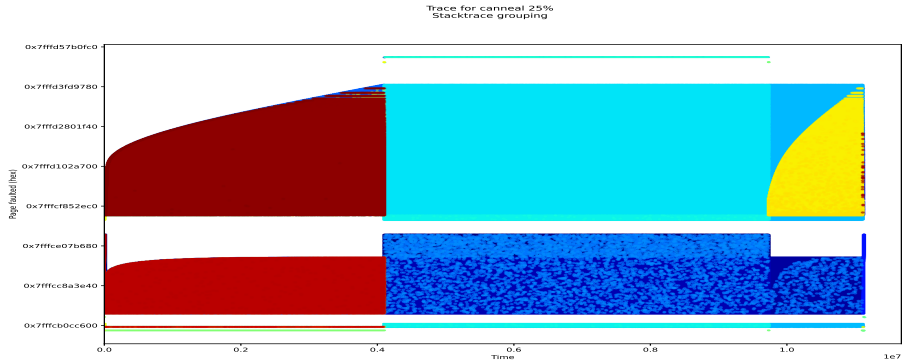


(c) 75%

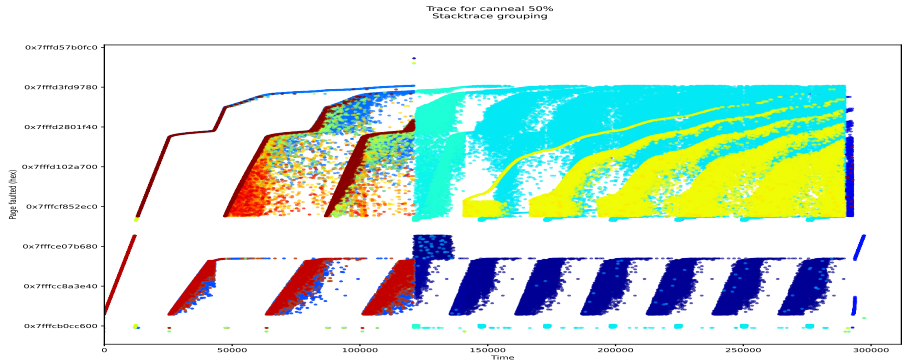


(d) 100%

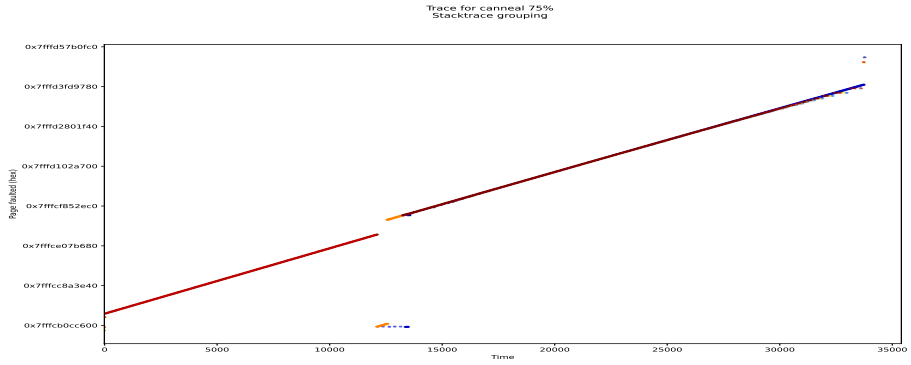
Figure 23: Bodytrack



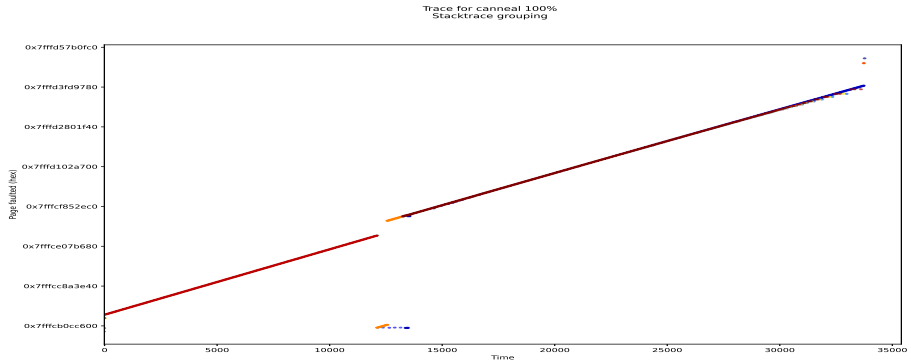
(a) 25%



(b) 50%

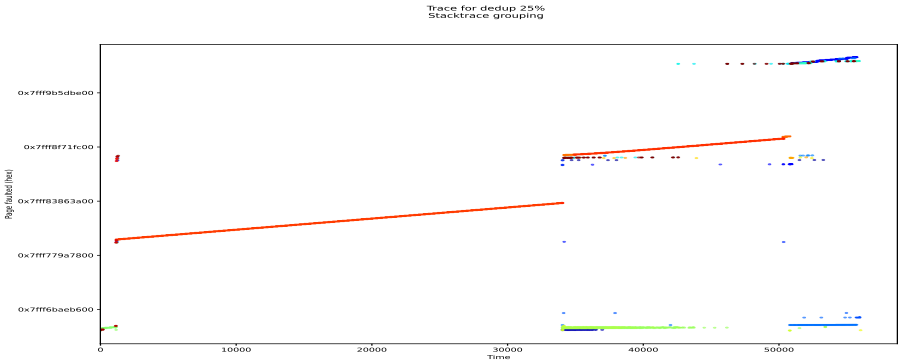


(c) 75%

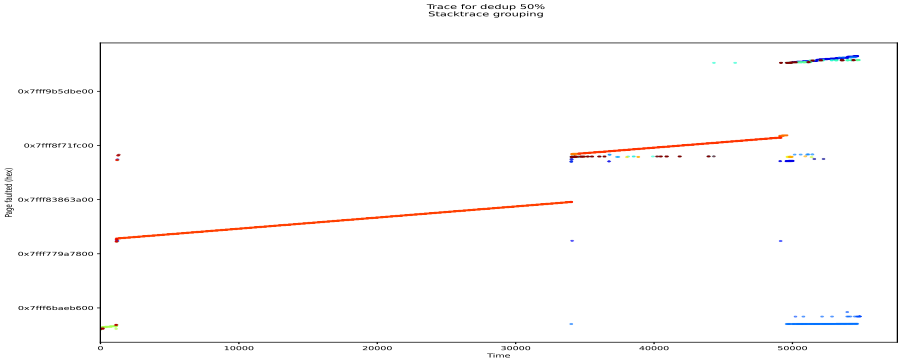


(d) 100%

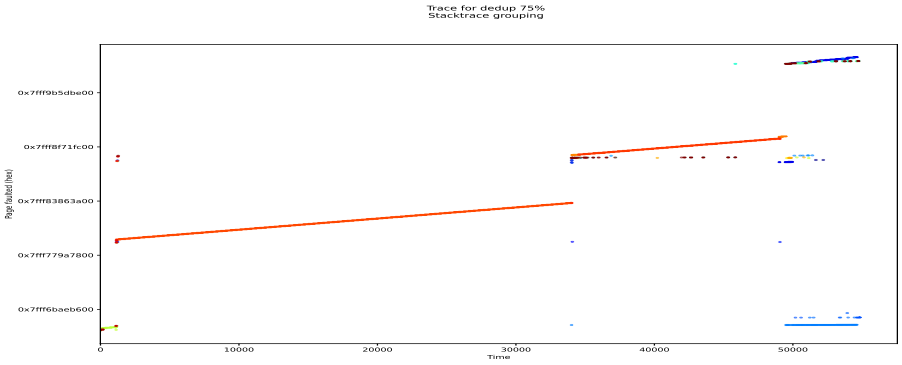
Figure 24: Canneal



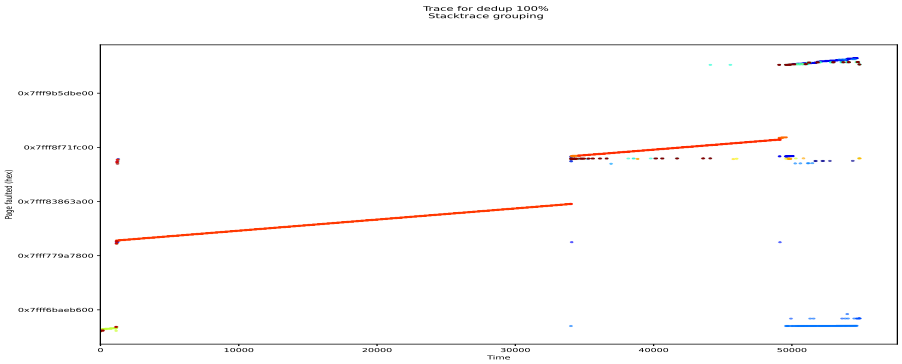
(a) 25%



(b) 50%

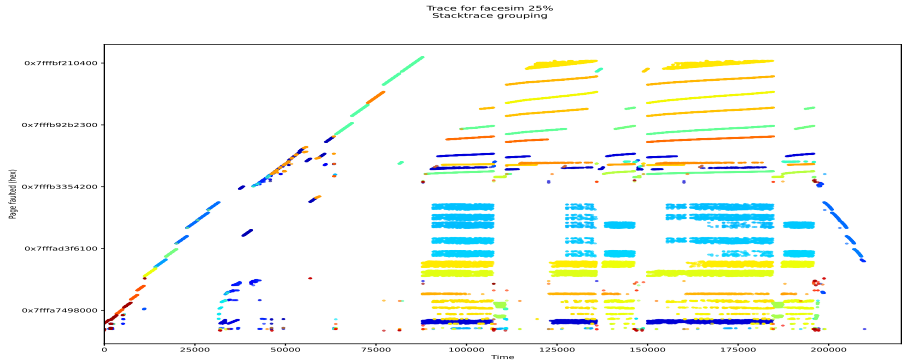


(c) 75%

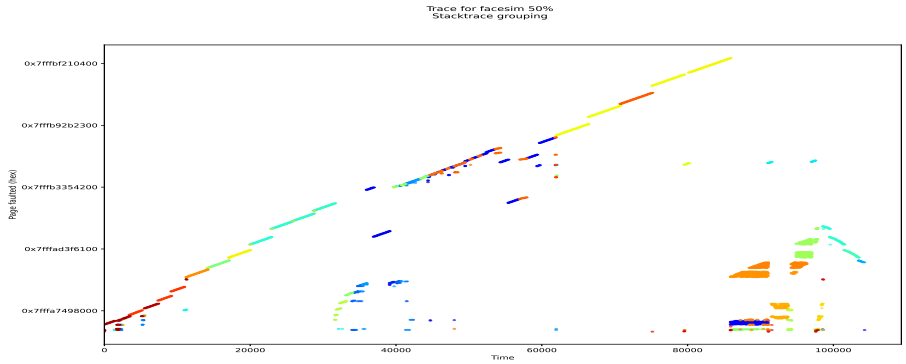


(d) 100%

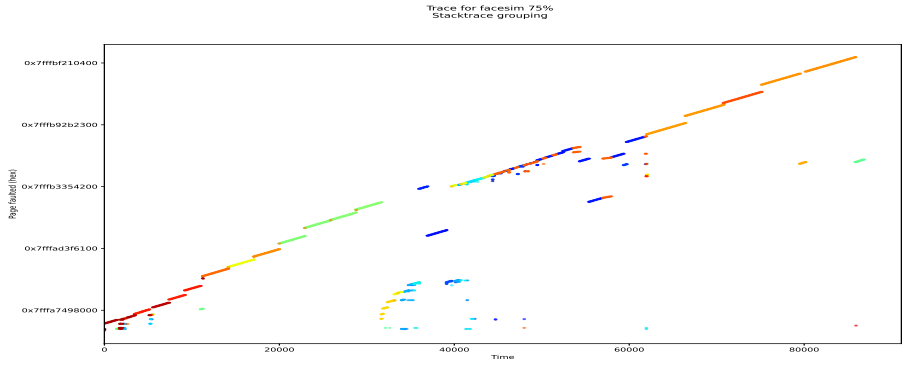
Figure 25: Dedup



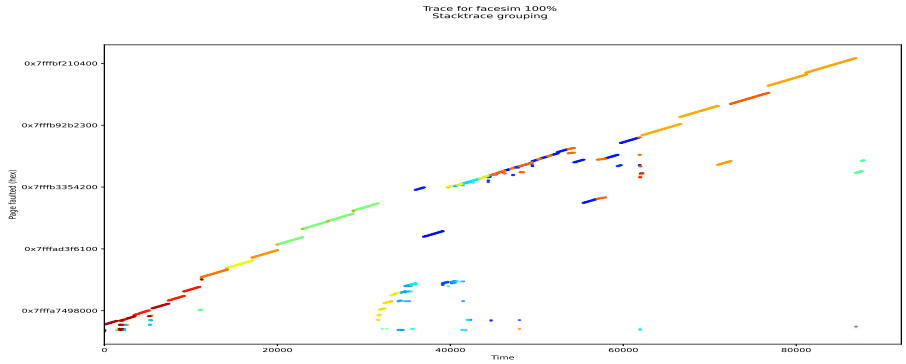
(a) 25%



(b) 50%

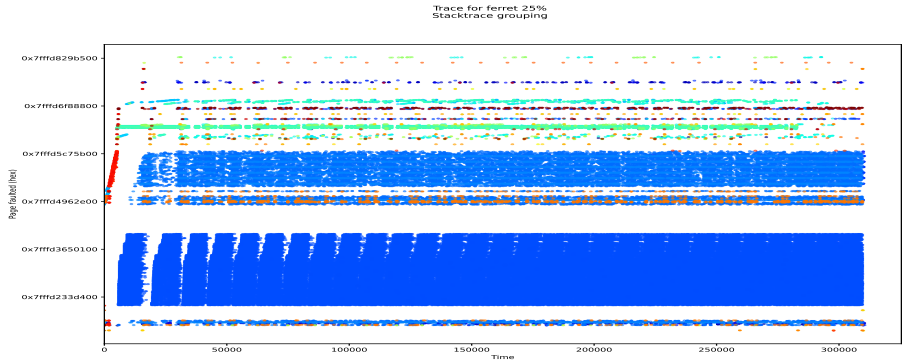


(c) 75%

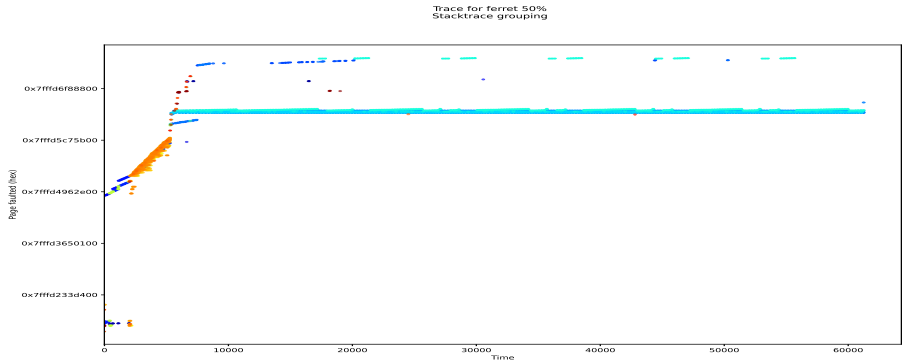


(d) 100%

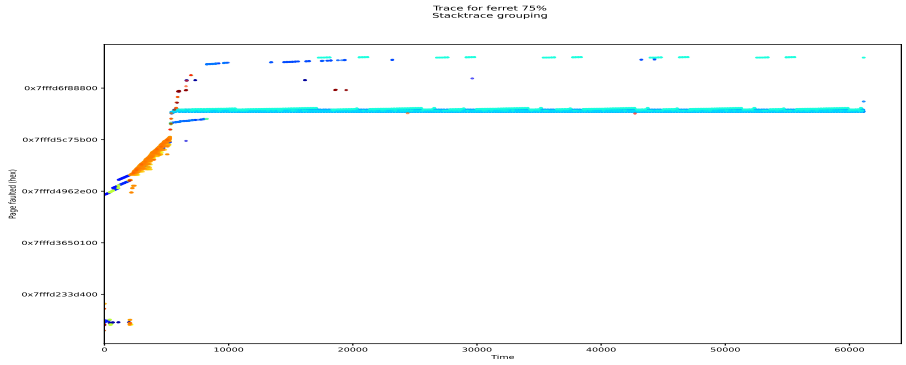
Figure 26: Facesim



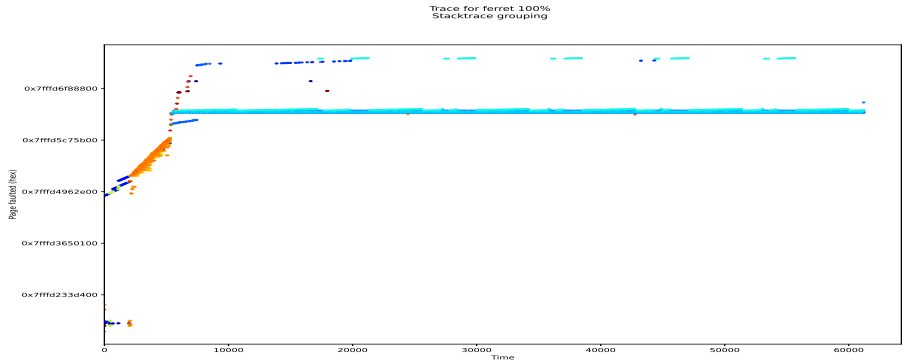
(a) 25%



(b) 50%

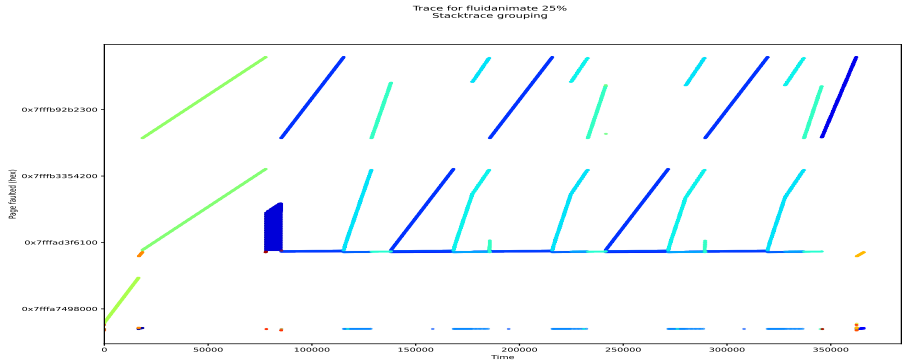


(c) 75%

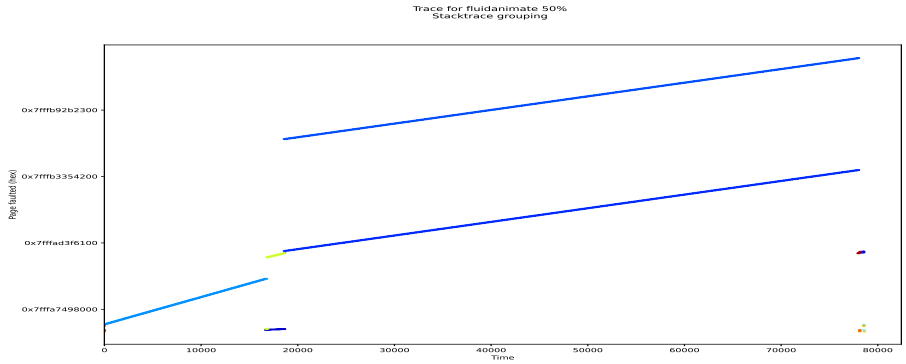


(d) 100%

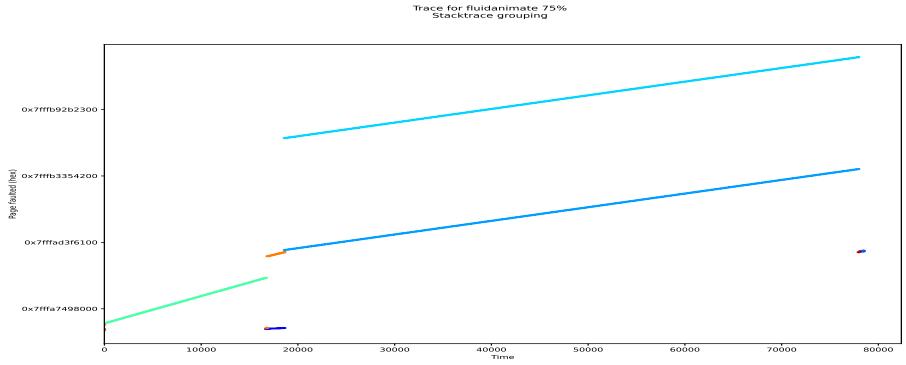
Figure 27: Ferret



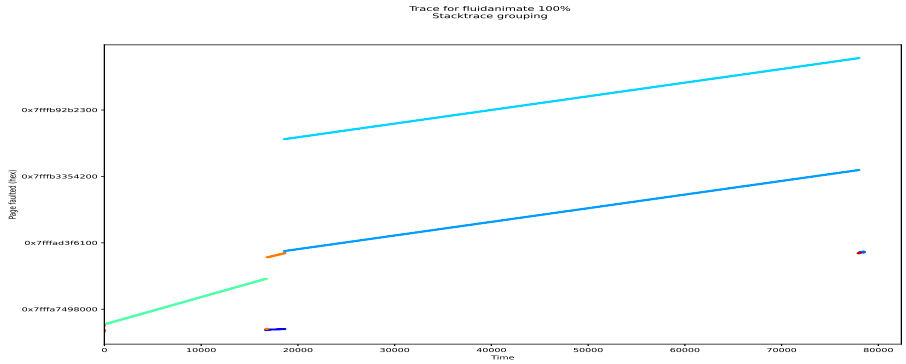
(a) 25%



(b) 50%

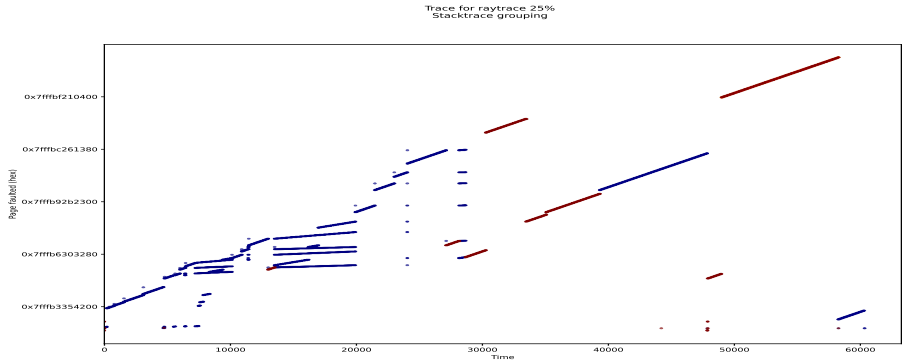


(c) 75%

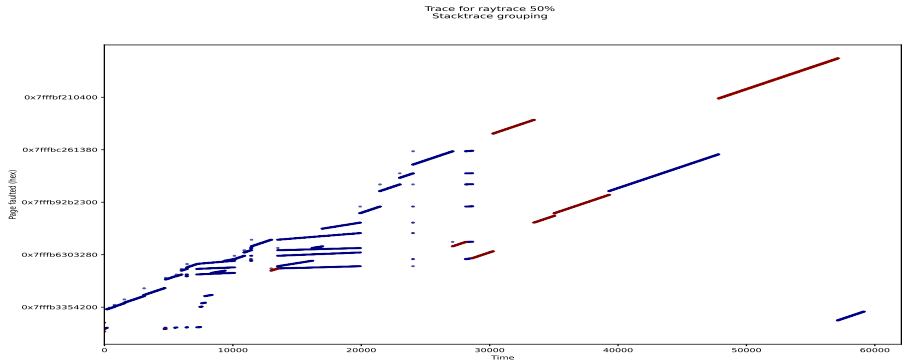


(d) 100%

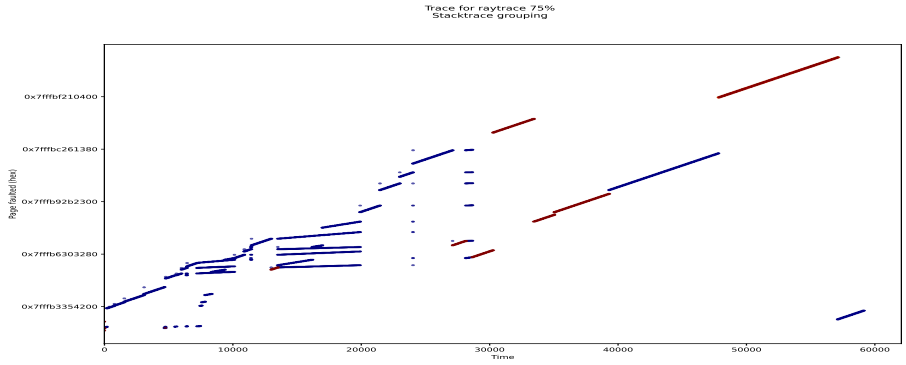
Figure 28: Fluidanimate



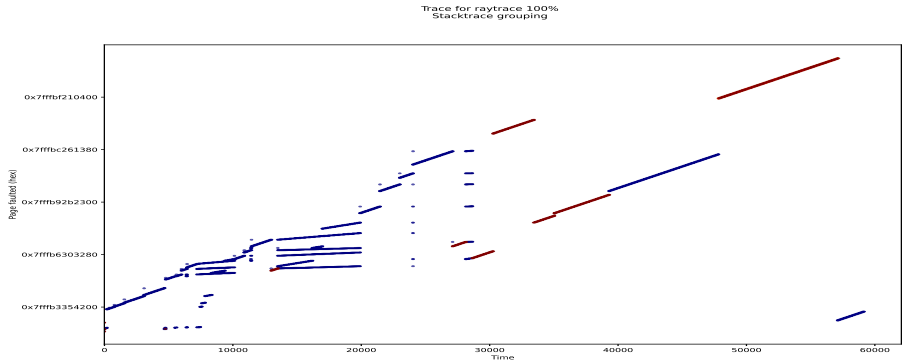
(a) 25%



(b) 50%

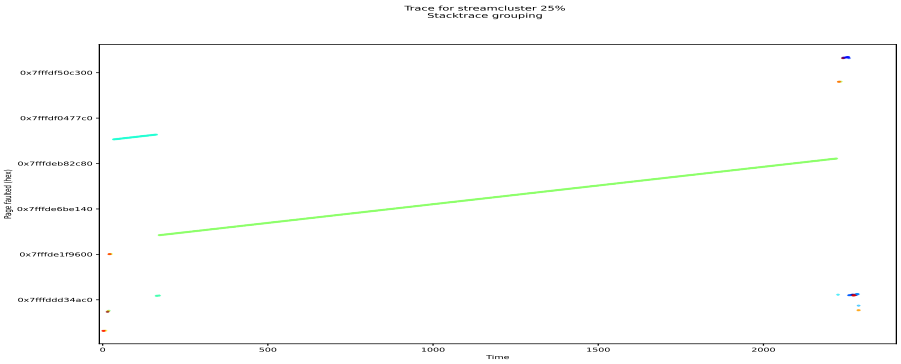


(c) 75%

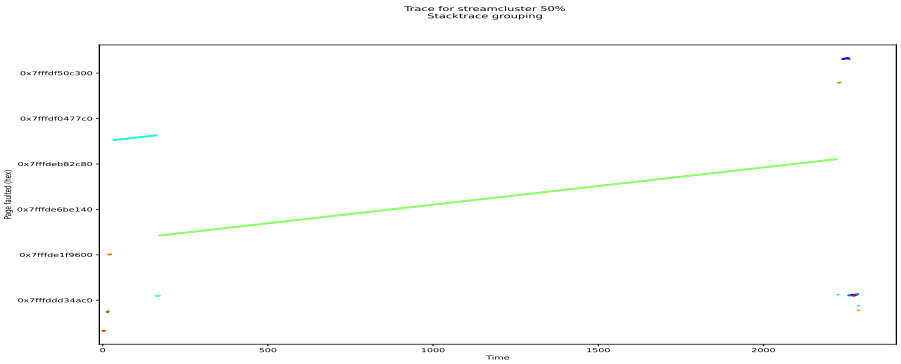


(d) 100%

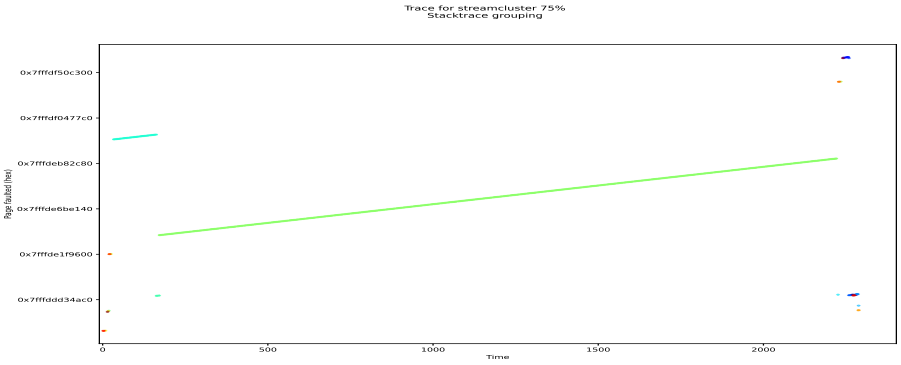
Figure 29: Raytrace



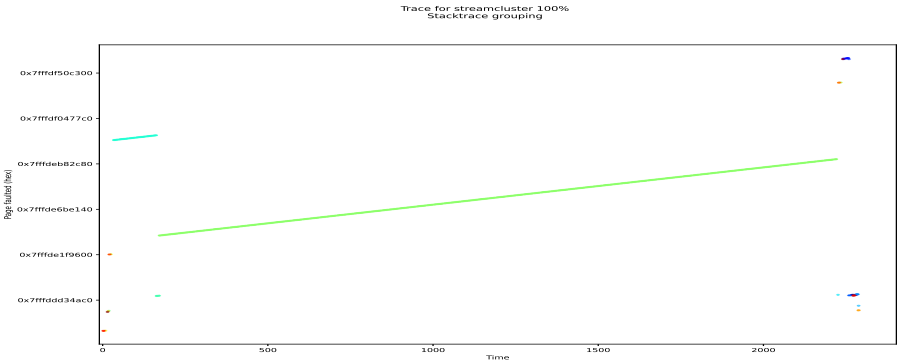
(a) 25%



(b) 50%



(c) 75%



(d) 100%

Figure 30: Streamcluster