

# Online Working Set Change Detection with Constant Complexity

The Cornerstone for Memory Management Algorithms in Scalable Systems

L J Gokul Vasani  
Nvidia  
Munich, Germany  
gokull@nvidia.com

Éder F. Zulian  
TU Kaiserslautern  
Kaiserslautern, Germany  
zulian@eit.uni-kl.de

Christian Weis  
TU Kaiserslautern  
Kaiserslautern, Germany  
weis@eit.uni-kl.de

Matthias Jung  
Fraunhofer Institute for Experimental  
Software Engineering (IESE)  
Kaiserslautern, Germany  
matthias.jung@iese.fraunhofer.de

Norbert Wehn  
TU Kaiserslautern  
Kaiserslautern, Germany  
wehn@eit.uni-kl.de

## ABSTRACT

In advanced scalable multiprocessor systems with multiple applications running concurrently, the optimal utilization of memory resources plays an essential role in achieving high system performance and energy efficiency. Locality of reference is one of the basis of memory management techniques. In operating systems, the memory management benefits from temporal and spatial locality when deciding which page shall be moved down the memory hierarchy to make room for a new page. However, current algorithms are oblivious to the moments when individual tasks transition to a new working set, which leads to suboptimal decisions. Furthermore, the eviction of idle pages from main memory is postponed to the last minute, which leads to system unresponsiveness and thrashing. In this context, memory locality reasoning based on the phase transition model is a central aspect to consider. We perceive that incorporating such a model in memory management algorithms provides the operating system with ways to trigger proactive page reclamation or compaction, to collect memory related statistics, and to trigger other crucial decision-making.

In this paper, we present for the first time a hardware-agnostic online working set change detection algorithm for each process in a system. The working set change is detected with incomplete knowledge of a task's memory access behavior. Our novel online algorithm perceives the occurrence of concavity in a program's lifetime function as an outlier and heuristically attempts to classify such from the normal observations. Furthermore, we implemented and evaluated our solution in the Linux kernel. It executes with  $O(1)$  time and space complexity. Evaluations using real-world data-intensive applications show that the algorithm detects the working set change with an average accuracy of 89% and F0.5-Score of 74%, adding negligible computation overhead of 0.506% in average.

## CCS CONCEPTS

- **Theory of computation** → **Caching and paging algorithms;**
- **Software and its engineering** → *Virtual memory; Main memory.*

## KEYWORDS

memory management, paging, locality of reference, working set, phase transition behavior, online detection

### ACM Reference Format:

L J Gokul Vasani, Éder F. Zulian, Christian Weis, Matthias Jung, and Norbert Wehn. 2021. Online Working Set Change Detection with Constant Complexity: The Cornerstone for Memory Management Algorithms in Scalable Systems. In *Proceedings of The International Symposium on Memory Systems (MEMSYS 2021)*. ACM, New York, NY, USA, 16 pages.

## 1 INTRODUCTION

The increasing capacity of the main memory, which is primarily based on DRAM, has favored data-intensive computations like Big data [33, 85, 105] and in-memory databases [1–3, 25, 70, 106]. At the same time, it has intensified DRAM-related challenges like increased power consumption due to DRAM refreshes [72] and possible data loss due to inadvertent system failures. Hence, optimal utilization of the memory resources by retaining only working set pages in the main memory while securing other pages in non-volatile storage or memory can be considered highly suitable for many of today's high performance scalable systems.

However, in largely adopted operating systems, such as Linux and other Unix-like derivatives, the current page reclamation approaches that evict idle pages from the main memory demands changes to keep pace with the evolution of applications in terms of memory access behavior. For instance, the decision-making process of which pages shall be evicted is oblivious to the working set transitions of individual tasks. Moreover, the execution of the page reclamation algorithm is postponed to the last minute, hence the eviction of cold pages from main memory. In practice, the operating system decides to trigger the page reclamation algorithms only when the memory utilization reaches a *Watermark* [18, 52, 77–79]. Watermarks correspond to certain thresholds with respect to memory utilization. They are computed as a fixed amount of free physical memory pages in the system. Typically, watermarks are very close to the point of memory exhaustion, which is the cause for retention of cold pages in the main memory [21, 98], regarded as memory pollution. Further, it leads to system unresponsiveness and out-of-memory panics [71, 86, 99, 104].

Reclamation algorithms are online approximated algorithms [17] that need to run periodically to attain precision in the page access rate sequence. However, the recurrent execution of reclamation algorithms at short intervals hinders the system performance. We perceive that a strategy to address the mentioned challenges follows the working set policy strictly, i.e., it keeps only the working set pages in main memory. To attain this strategy, a page reclamation algorithm needs to be made aware of (1) the current working set pages required by each of the tasks in execution, and (2) the moments when a task transitions to a new working set. The latter can be employed to optimize the main memory utilization by evicting cold pages at *essential intervals*. Moreover, the methodical migration of former hot pages down the memory hierarchy (from main memory to non-volatile memory) maintains only the required pages in the main memory and allows for improving the energy efficiency in the memory subsystem and reducing data loss due to inadvertent system failures. Further, this approach can decrease the probability of a system reaching unresponsive states due to memory exhaustion and trashing.

The purpose of this work, therefore, is to present a novel hardware-agnostic algorithm able to detect at execution time when a task switches to a new locality interval i.e., detect the transitions between two different working sets. For that, we exploit the intuition that a reasonably designed program will exhibit relatively deep concave lifetime curves during its prime locality's holding state. This phenomenon generates a relatively long time gap between the page faults that occurred while transitioning from one locality to another (cf. Section 2). Compared to other observations this is a deviation. Our solution targets to identify such outliers.

Furthermore, we foresee that the moments in which working set transitions occur can provide the operating system with opportunity to trigger proactive page reclamation, trigger page compaction, implement dynamic watermarks, plainly rotate the pages in the reclamation queue [52, p. 170] to achieve comprehension of the current working set pages and by that increase the accuracy of the reclamation algorithm, collect memory statistics, trigger other crucial decision-making regarding resource utilization. Moreover, the related software components can be adapted to become event-triggered, eliminating polling and fixed interval probing. These are the topics we plan to treat in future work.

We implemented the proposed algorithm in the Linux kernel along with the instrumentation and other software artifacts necessary to evaluate it. We perform a thorough evaluation of the proposed solution using multiple real-world relevant data-intensive applications, such as MongoDB and Memcached. In order to validate the correctness of our evaluation technique we perform a trace-based analysis which is often used for locality modeling. Evaluations show that the proposed solution detects the working set change with an average accuracy of 89% and F0.5-Score of 74%, adding negligible computation overhead of 0.506% in average.

In summary, the major contributions of this work are as follows:

- We propose a software-only solution in the form of an online algorithm that solves the challenge of identifying the essential intervals in which a task shifts to a new working set.
- When detailing the basic blocks of our solution, we provide an elaborate reasoning of our ideas to implement the proposed

algorithm that will detect locality changes of a task in a time-sharing system.

- We introduce formal definitions for locality noise and Inter Page Fault Virtual Time (IPFVT). IPFVT is a hardware-agnostic metric that serves as one of the foundations for working set change detection in time-sharing systems.
- We formally model and define the phase transition behavior of a program [39] into a Deterministic Finite State Automaton (DFA).
- We designed and implemented our algorithm as a transition function for the phase transition DFA. The algorithm introduces negligible computation overhead, needing a fixed number of conditional audits, uttering no iterative behavior with  $O(1)$  time and space complexity, making its integration feasible into state-of-the-art memory management software.
- We provide novel and detailed evaluation by precisely defining the ground truth and reducing the results into a binary classification problem to measure the various accuracy metrics of the proposed algorithm.
- We perform a thorough evaluation of the proposed solution using multiple real-world data-intensive workloads to demonstrate the viability of its integration in the Linux kernel and its accuracy.

The rest of this paper is structured as follows: In Section 2 we briefly review the necessary background. Section 3 describes system model and notations. We define and reason the principles that form the basic blocks of the algorithm in Section 4. In Section 5 we detail our novel online algorithm along with the definition of the methods based on the principles introduced in the Section 4. In Section 6 we establish the ground truth by designing an optimal offline algorithm whose results are used as a standard measuring stick to compare the various parameters of our approximated online algorithm. Section 7 presents various experiments conducted using real-world databases and well-known benchmarking tools to measure different metrics to understand the proposed algorithm's practicality and accuracy. The same section reports the baseline comparison of our proposed algorithm against other well-known access pattern-based phase-transition algorithms using marked synthetic traces. Further, the synthetic trace evaluation is performed to understand our algorithm's theoretical accuracy and corroborate our evaluation procedure's accuracy over real-world applications. Section 8 discusses related work and Section 9 concludes the paper.

## 2 BACKGROUND

The reasoning of a program's memory access behavior starts with capturing a program's machine independent memory trace named *reference string*  $\omega$  [28] such that:

$$\omega = r_1, r_2 \dots r_t \in \mathbb{N}, t \geq 1.$$

where  $t = 1, 2, 3, \dots$  represents the memory access instants in a process's discrete time units. The  $r_t$  is a relation defined by  $r_t = i$  implies the page  $i$  is referenced at the  $t^{th}$  instance.

If the captured reference string of any sequential program is examined, the program will tend to cluster the memory referencing to a small set of its memory pages in both temporal and spatial aspects, termed *locality of reference* [4, 41, 101].

Of the many [5, 28, 38, 40, 47, 93, 95], the *working set model* ( $W$ ) [38, 40] represents a program's locality in single relation. Formally,

if  $\omega$  is the reference string generated by a program then the working set can be defined as  $W(t, \tau)$  such that:

- $t$  is the moment at which  $\omega$  is observed
- $\tau$  is the window length backwards from  $t$

Intuitively, it is a set of pages referenced in the time interval of length  $\tau$  preceding time  $t$ . The working set idea is based on the observation that the pages referenced in the backward window were likely to be used again in the immediate future [41].

During the days of scarce main memory, the working set as a memory policy [40] (a program's resident set size is at least the working set size) dictated the ground rule to avoid thrashing [58, 63]. The working set policy for memory management states that a task is qualified to be in the run queue only if its working set is loaded in main memory [39].

The phase-transition model [39, 49, 53] reasons the empirical observation that any gross program's memory access behavior can be seen to be in one of two states: transition or in-phase (holding) [40, 65]. On entering a new locality, a program at first enters the transition state, during which the program exhibits page faults at close successive intervals. Then the program enters the holding state once the needed pages for that locality are in main memory, where it stabilizes and stops faulting for a relatively long time interval. In holding state the length of the reference string determines the degree of locality, and each program tends to emit varying degrees of locality at different stages of its execution.

The *Bounded Locality Interval* (BLI) [12, 24, 68, 74] solves the problem of identifying distinctive phases by exploiting the observation of nested localities in a program where the larger (prime) localities are composed of smaller ones. This showed that  $\tau$  in  $W$  does not need to be a constant value.

The Lifetime Function [42, 43] measures the degree of locality defined as the mean number of page references between the page faults. This function will exhibit a relatively deeper concave curve (knee characteristics) [94] between a program's transition to the next working set. The depth of the curve determines the degree of the locality.

### 3 SYSTEM MODEL AND NOTATIONS

The system is assumed to have an execution unit and is time-shared among many independent sequential execution entities called *tasks*.

A running task executes on its virtual address space  $N$  divided into contiguously addressed  $c$  pages, each of size  $z$  and defined by  $N = \{1, 2, \dots, c\}$ . The system's main memory space  $P$  is divided into  $n$  contiguous locations named as page frames, each of size  $z$  and defined by  $P = \{1, 2, \dots, n\}$  such that  $1 \leq n \leq c$ .

$C_t : N \rightarrow P \cup \{\emptyset\}$  is an one-to-one page map at time  $t$  such that:

$$C_t(x) = \begin{cases} y, & \text{if } x \text{ resides in } y \in P \text{ at time } t \\ \emptyset, & \text{if } x \text{ is missing from } P \text{ at time } t \end{cases} \quad (1)$$

where  $x \in N$  represents the page in the virtual address space of the corresponding task and  $y \in P$  represents the physical page frame. The mapping of a page to its corresponding page frame is arbitrated by a memory translation hardware that resides between processor and the main memory.

When the running task accesses a virtual address  $a$  that can also be represented as  $a = xz + w$  where  $x$  is the corresponding

virtual page number and  $0 \leq w < z$ , then the memory translation hardware maps this to a corresponding physical memory address using the computation

$$B = C_t(x)z + w \quad (2)$$

In demand paging based system, if  $C_t(x)$  is  $\emptyset$ , then the hardware generates an interrupt (page fault) to the operating system. The operating system resolves the page fault by assigning the faulted page with the corresponding page frame. The operating system work-conserves [22, p. 70] the system by relinquishing the faulted task to another ready task till the page fault is resolved.

The page fault time is a function  $p : \mathbb{Z}^* \rightarrow \mathbb{R}^+$  that takes the instance index of the page fault occurrence and outputs the wall clock time at which the page fault occurred. The function  $p$  generates an ordered sequence  $\gamma \in \mathbb{R}^+$  of page fault time for each process and is a machine-dependent order, and represented as:

$$\gamma = g_0, g_1, g_2, \dots, g_{t-n}, \dots, g_t \in \mathbb{R}_+^t, t \geq 0 \quad (3)$$

where,  $g_i = p(i)$  such that  $i \in \mathbb{Z}^*$  is the instance index of the page fault and  $g_i$  represents the wall clock time of the corresponding page fault instance index  $i \in \mathbb{Z}^*$ .

Assume, the system would run a single task with no preemption; then the inter page fault time is the time elapsed between the two consecutive page fault which is a relation:

$$d_t = \{g_t - g_{t-1} | g \in \gamma\} \quad (4)$$

The  $\delta$  is the ordered sequence of inter page fault time defined by:

$$\delta = d_1, d_2, \dots, d_{t-n}, \dots, d_t \in \mathbb{R}_+^t, t \geq 1 \quad (5)$$

where the discrete unit  $t = 1, 2, \dots$  represents the ordered index in correspond to  $\gamma$ 's sequence order.

## 4 ALGORITHM - BASIC BLOCKS

In this section we define the principles of the basic blocks of the proposed algorithm: **(i)** extend the inter page fault time to support time-sharing systems, **(ii)** model a task's distinctive locality changes into a Deterministic Finite state Automaton (DFA), which applications can probe to check the current state of a task, **(iii)** define local maxima of an IPFVT sequence, which allows for identifying a transition to a new working set, **(iv)** establish the basis for locality noise that might cause locality jitter, and **(v)** define the sliding window computation that is the basis for the transition detection in the online algorithm. The principles described in this section will be used in Section 5 to design the online algorithm.

### 4.1 Inter Page Fault Virtual Time (IPFVT)

In time-sharing systems, a task might be preempted often. Preemption induces additional wall clock time to a task's execution time and so to the inter page fault time. We name this additional time as scheduler noise  $S$ .

**Definition 1.** Let  $d_k \in \delta$  represent one inter page fault time instance of the task  $T$  occurred within the interval  $(t, t + d_k]$ ; if  $s_i$  represents one instance of the preemption time within the interval  $(t, t + d_k]$  and  $p$  such preemptions occurred within the same interval, where  $p \geq 0$ ; then the scheduler noise  $S_p$  for the interval

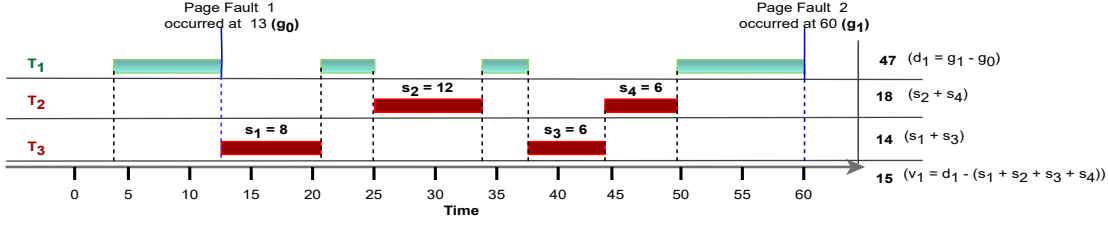


**Figure 1: Exemplification of IPFVT.**  $g_0$  and  $g_1$  are the timestamps of two consecutive page faults of task  $T_1$ , while  $s_1$ ,  $s_2$ ,  $s_3$ , and  $s_4$  represent the time spans during which  $T_1$  was preempted between the page faults.

$(t, t + d_k]$  can be calculated as:

$$S_p = \sum_{i=0}^p s_i \quad (6)$$

and, the inter page fault virtual time  $v_k$  for the  $d_k$  instance is defined by the relation:

$$v_k = \{d_k - S_p | d_k \in \delta\}. \quad (7)$$

Figure 1 exemplifies one instance of the  $v$  computation. The Definition 1 is abstracted as a function  $Compute\_ipfvt : \mathbb{R}^+ \rightarrow \mathbb{R}^+$  that computes IPFVT from the page fault time  $g_n \in \gamma$  generated at time 0 (cf. Section 3).

Each process will generate an ordered sequence of *inter page fault virtual time*  $V$  represented as:

$$V = v_1, v_2, \dots, v_{t-n}, \dots, v_t \in \mathbb{R}^+, t \geq 1 \quad (8)$$

where  $t = 1, 2, \dots$  represents the ordered index in correspond to  $\delta$ 's sequence order. Let the function  $I : V \rightarrow \mathbb{Z}^+$  return the ordered index  $i \in \mathbb{Z}^+$  of the  $v_i \in V$ . Let the relation  $\preceq$  represent the order in a sequence, i.e.,  $a \preceq b$  means  $a$  occurred before  $b$  in its corresponding sequence, then the distance between two  $v_k \in V$  is a function  $B : V \times V \rightarrow \mathbb{Z}^+$  computed using:

$$B(v_i, v_j) = \{I(v_j) - I(v_i) | v_i \preceq v_j \wedge v_x \in V \text{ where } i, j \in x\} \quad (9)$$

Using IPFVT as a metric enabled us to derive a solution regardless of the practical hardware constraint that, for a given page, a hardware memory management unit only sets a reference bit in the page table on first access and will not provide any page access pattern sequence [7, 31, 73].

## 4.2 Modeling Locality of a Program

The locality behavior of a task can be represented by the DFA  $\mathcal{A}$  presented in Figure 2.

**Definition 2.**  $\mathcal{A}$  is a 5-tuple  $\mathcal{A} = (Q, q_0, F, \Sigma, \sigma)$ , consisting of a set of finite states  $Q$ , an initial state  $q_0 \in Q$ , an accepting state  $F \in Q$  and a finite set of input symbols  $\Sigma$  and a transition function  $\sigma : Q \times \Sigma \rightarrow Q$ , where:

- $\{Q\} = \{Init, Holding, Transition, Exit\}$
- $q_0 = \{Init\}$
- $F = \{Exit\}$
- $\{\Sigma\} = \{\emptyset, Up, Down, Term\}$
- $\sigma = DetectLocalityChange$ .

It is sufficient to represent a task's memory allocation behavior with four states. The *Init* state represents the task creation, during

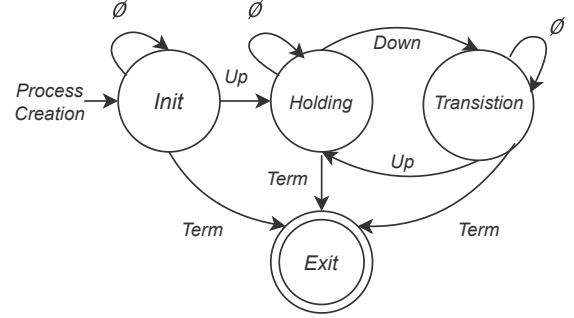


**Figure 2: DFA Representing Locality Behavior of a Task**

which the system initializes, allocates and loads the initial pages for the task. It also calibrates the initial expectations of the algorithm. Once initialized the task will transit between *Holding* and *Transition* states (cf. Section 2). When the *Exit* state is reached all pages allocated for the task are released and the task departs from the system. The function *DetectLocalityChange* (cf. Section 5) is responsible for state transitions. It computes the next state based on the current state and the input symbol.

## 4.3 Local Maxima

To make a reasonable detection of the outermost phases  $\varphi$ , we use IPFVT  $V$  (cf. Section 4.1) as the primary factor of identification. Suppose a program is executed in advance to capture the complete sequence of the IPFVT  $V$ , we will notice that the first IPFVT value while entering the transition state will be relatively a higher single point value than its neighbors. Such a high single point value is due to a significantly longer reference string during the stable execution of the corresponding program between the locality transition faults compared to its neighboring page faults. This behavioral observation of the highest single point value in a sequence of observations is the local maxima.

**Definition 3.** Consider  $V' = v_i, \dots, v_{i+k}$  is a contiguous subsequence of  $V$ , such that  $0 < i < t$  and  $0 \leq k \leq t$  and it starts at position  $i$  and ends at position  $i + k$  then,  $V'$  could be split into

$$V' = (v'_B)v_m(v'_A), \quad (10)$$

such that,  $v'_B = \langle v_i \dots v_{i+m-1} \rangle \cup \emptyset$  and  $v'_A = \langle v_{i+m+1} \dots v_{i+k} \rangle \cup \emptyset$  where,  $i < m < i + k$ ; The element  $v_m \in V'$  is a local maxima iff:

$$(\forall v_b \in v'_B (v_b < v_m)) \wedge (\forall v_a \in v'_A (v_a < v_m)). \quad (11)$$

The sequence  $V$  will generate a subsequence [56] of local maximas  $M$  represented by:

$$M = m_1, m_2, m_3, \dots, m_t \text{ such that } t \geq 1 \quad (12)$$

$t = 1, 2, 3, \dots$  represents the ordered sequence of occurrence in the corresponding  $v_i \in V$ .

The function  $\mathcal{R} : M \rightarrow V$  will reverse map the given  $m_i \in M$  to its corresponding  $v_i \in V$ , then the distance between two maximas can be defined as  $\mathcal{D} : M \times M \rightarrow \mathbb{Z}^+$  and computed using :

$$\mathcal{D}(m_i, m_j) = \{I(\mathcal{R}(m_j)) - I(\mathcal{R}(m_i)) | m_i \triangleleft m_j \wedge m_x \in M \text{ where } i, j \in x\}. \quad (13)$$

The distance operator  $\mathcal{D}$  emits the IPFVT count between two local maximas.

#### 4.4 Locality Noise

An implicit characteristic of the bounded locality intervals is the hierarchical nature of localities, i.e., localities at lower levels are subsets of a higher-level locality. The BLI reasons that the phases of a higher degree locality hold a much extended locality than its subsets, and they have a high possibility of exhibiting non-overlapping behavior with their neighboring phases. Empirically, such behavior in the higher degree of locality is due to stable execution of the program relative to its short-lived lower degree of locality.

Considering the aforementioned a locality change detection algorithm has to target the identification of the outermost phase. However, the objective function that detects only the outermost phases might unintentionally identify the localities of a lower degree. Such an inadvertent detection has to be discriminated against normal observation and treated as locality noise  $\eta_o$  of a program.

#### 4.5 Sliding Window Computation

We use moving average based computation to record only the past's essence in our stream-based online algorithm, which is defined as:

**Definition 4.** The moving window  $\mathcal{F}$  is a 5 tuple  $\mathcal{F} = (C, l, S, \bar{a}, r)$ , consisting of  $C = \{\text{Initial}, \text{Drifting}\}$  a set of states the window will exhibit,  $l \in \mathbb{N}^+$  is the length of the window,  $S \in \mathbb{R}^+$  is the latest  $l$  elements of the given sequence and  $r$  holds the at present computed result of the function  $\bar{a}$ , where a window is said to be in *Initial* state when the sequence  $S$  is yet to attain the length  $l$  otherwise, it is in *Drifting* state, and  $\bar{a} : S \times \mathbb{N} \rightarrow \mathbb{R}^+$  is computed as:

$$\bar{a}(s_n, l) = \begin{cases} s_n + s_{n-1} + \dots + s_0, & n < l \\ \frac{(s_n + s_{n-1} + \dots + s_{n-(l-1)})}{l}, & \text{otherwise} \end{cases} \quad (14)$$

where,  $s_n \in S$  and  $n \in \mathbb{N}$ .

We use IPFVT as the window metric as this makes the algorithm platform-independent. Further, tuning such a window is much precise and could be easily automated. For the convenience of readability we use the dot operator "." to access the elements of the window tuple, for example, if  $w_i \in \mathcal{F}$  then,  $w_i.l$  represents the length of the window  $w_i$ .

### 5 THE ONLINE ALGORITHM

In this section, we (i) define an online method to differentiate the locality noise from the primary locality, (ii) a method to detect the locality transition of a task, (iii) a method to detect a task's holding state, and finally (iv) assemble the sub-algorithms explored

in (i) to (iii) to form the final algorithm that determines the locality state of a task. The online algorithm does not have the complete

---

#### Algorithm 1: Determining Locality Noise and Adaptive Elements

---

**Input:**  $v_t \in V_o$  - Generated at time 0.  
 $n$  - Current  $v_t$  count.  
 $s_c$  - The current state of the executing program.  
 $s_p$  - Immediate previous state.  
 $c$  - Counter for  $\eta_o$  determination in transition.  
 $w_s$  - Short term moving window.  
 $b$  - Noise band.  
 $p$  - Placeholder for temporary computation.  
 $\eta_t \leftarrow 0$   
**return value:**  $\eta_t \in \eta_o$  if true, else 0.  
**Algorithm** Determine  $\eta_o()$   
**if**  $w_s.C = \text{Initial}$  **then**  
     $p \leftarrow \max(v_t, p)$   
**end**  
**else if**  $n = w_s.l$  **then**  
     $\eta_t \leftarrow \max(v_t, p)$   
**end**  
**else**  
    **if**  $s_c = \text{Holding}$  **then**  
        **if**  $s_p = \text{Transition}$  **then**  
             $p \leftarrow \infty$   
        **end**  
        **if**  $v_t \geq b$  **then**  
             $p \leftarrow \min(p, v_t)$   
        **end**  
    **end**  
    **else if**  $s_c = \text{Transition or } s_c = \text{Init}$  **then**  
        **if**  $s_p = \text{Holding}$  **then**  
            **if**  $p \neq \infty$  **then**  
                 $\eta_t \leftarrow p$   
            **end**  
             $p \leftarrow 0$   
             $c \leftarrow 0$   
        **end**  
        **if**  $c \geq w_s.l$  **then**  
             $\eta_t \leftarrow \max(v_t, p)$   
             $c \leftarrow 0$   
        **end**  
        **else**  
             $p \leftarrow \max(v_t, p)$   
             $c \leftarrow c + 1$   
        **end**  
    **end**  
**end**  
**return**  $\eta_t$

---

information of the sequence  $V$  rather, it is a live stream. To elaborate, the Section 4.3 dissected the sequence  $V$  into  $v_B$ ,  $v_m$ , and  $v_A$ . During the online computation the  $v_m$  has to be determined with only  $v_B$  and the knowledge of  $v_A$  is not available. We assume  $v_A$  will tend to infinity. We define this stream based incomplete sequence of  $V$  as  $V_o$ .

#### 5.1 Self Adaptive Noise Band

Though the locality noise  $\eta_o$  is a deviation from the normal behavior, it failed to qualify as an element that will cause the change in the primary locality. The addition of only such deviated elements will numerically raise the band. In order to mitigate this challenge, the adaptive noise band algorithm (cf. Algorithm 1) reduces the expectations by normalizing the noise band  $b$  with adaptive elements which is based on the program's behavior. This approach will reduce the probability of failing to detect prime localities with smaller maximas.

**Algorithm 2: Hypothesis Testing Algorithm**


---

**Input:**  $v_t \in V_o$  - Generated at time 0.  
 $c$  - Counter for "near future" holding determination.  
 $b$  - Noise band.  
 $w_s$  - Short term moving window.  
 $w_l$  - Long term moving window.  
 $v_p$  - presumptive element to be  $m \in M$ .  
 $\eta_t \leftarrow 0$  - A value is assigned if  $v_p \in \eta_o$  or  $v_t \in \eta_o$ .

**output:** 0 - If  $v_p$  is  $m \in M$ .  
1 - If  $v_t$  or  $v_p$  is locality noise.  
2 - If  $v_t$  is speculated or is not valid.  
 $\emptyset$  - If nothing to test.

**Algorithm HypothesisTest ()**

```

if  $w_s.C = \text{Initial}$  then
  return  $\emptyset$ 
end
if  $v_t > b$  then
  if  $v_p = \emptyset$  then
     $v_p \leftarrow v_t$ 
     $c \leftarrow 0$ 
    return 2
  end
  else if  $v_p \neq \emptyset$  and  $v_t > v_p$  then
     $\eta_t \leftarrow v_p$ 
     $v_p \leftarrow v_t$ 
     $c \leftarrow 0$ 
    return 1
  end
  else if  $v_t < v_p$  then
     $\eta_t \leftarrow v_t$ 
    return 1
  end
end
if  $v_p \neq \emptyset$  then
  if  $c \geq w_s.l$  then
     $v_p \leftarrow \emptyset$ 
     $c \leftarrow 0$ 
    return 0
  end
   $c \leftarrow c + 1$ 
end
return  $\emptyset$ 

```

---

The algorithm *determining\_* $\eta_o$  (Algorithm 1) returns (i) the maximum of the elements encountered during the *Init* state, (ii) the minimum value among the elements that deviated from  $b$  during the *Holding* state and (iii) the maximum value encountered per  $w_s.l$  during the *Transition* state. The noise band  $b$  will be the average of the adaptive elements identified by the algorithm *determining\_* $\eta_o$  along with the elements deemed to be locality noise  $\eta_o$  by the hypothesis testing algorithm (cf. Section 5.3). This concise representation reduces the space and computation complexity in managing the locality noise elements.

## 5.2 Transition Detection

We use two moving windows,  $w_s \in \mathcal{F}$ , and  $w_l \in \mathcal{F}$ , per task as per the Definition 4 to determine the working set transition, where  $w_s.S$  and  $w_l.S$  represent the same sequence  $V_o$  generated by the corresponding task. If  $D_s$  is a sequence generated using the distance operator  $\mathcal{D}$  (cf. Section 4.3), from which we define basic window length  $L$  as  $L = \min(D_s)$  of a task. Then,  $w_s.l = L$  and  $w_l.l = 2 \cdot L$ . Choosing  $L$  as  $\min(D_s)$  makes a point that  $w_s$  is smoothed before it reaches the first holding state and  $w_l$  is smoothed before it reaches the second consecutive holding state.

This approach provides a platform independent and hardware agnostic window tuning semantic. The smaller length window  $w_s$  tends to fluctuate faster to the recent stream trends, while the longer

**Algorithm 3: Online Phase Detector**


---

**Input:**  $g_n \in \gamma$  - Generated at time 0.  
 $c_g$  - Current  $g_n$  count, set by the OS.  
 $b$  - Noise band.  
 $w_s$  - Short term moving window.  
 $w_l$  - Long term moving window.  
 $s_c$  - The current state of the executing program.  
 $s_p$  - The immediate previous state.  
 $c_h$  - Counter to determine "near future" holding.  
 $c_b$  - Counter for noise band determination.  
 $v_p$  - Hypothesis testing value.  
 $\eta_t$  - Phased Noise.  
 $tmp$  - Placeholder for temporary computation.

**output:**  $\emptyset$  - If no change in state.  
 $Up$  - If the task transitioned to Holding state.  
 $Down$  - If the task transitioned to Transition state.  
 $Term$  - If the task is terminating.

**Algorithm DetectLocalityChange()**

```

return_state  $\leftarrow \emptyset$ 
 $v_n \leftarrow \text{Compute\_ipfvt}(g_n)$ 
 $s_p \leftarrow s_c$ 
if  $c_g \geq 1$  then
   $n \leftarrow n + 1$ 
  if  $n = w_s.l$  then
     $w_s.C \leftarrow \text{Drifting}$ 
  end
  if  $n = w_l.l$  then
     $w_l.C \leftarrow \text{Drifting}$ 
  end
end
else
   $s_c \leftarrow \text{Init}$ 
   $s_p \leftarrow \text{Init}$ 
   $n \leftarrow 0$  //Initialize the  $v_t \in V_o$  count.
   $v_p \leftarrow \emptyset$ 
   $b \leftarrow 0$ 
end
 $w_s.r \leftarrow w_s.\bar{a}(v_n, w_s.l)$ 
 $w_l.r \leftarrow w_l.\bar{a}(v_n, w_l.l)$ 
if  $s_c = \text{Transition}$  or  $s_c = \text{Init}$  then
  ret  $\leftarrow \text{HypothesisTest}(v_n, c_h, b, w_s, w_l, v_p, \eta_t)$ 
  if ret = 1 then
     $b \leftarrow \text{reCompute\_noiseband}(b, \eta_t)$ 
  end
  else if ret = 0 then
     $s_c \leftarrow \text{Holding}$ 
    return_state  $\leftarrow Up$  //changed to holding state.
  end
end
if  $T(s_c, W_s, W_l) = 3$  then
   $s_c \leftarrow \text{Transition}$ 
  return_state  $\leftarrow Down$  //changed to transition state.
end
 $\eta_t \leftarrow \text{Determine\_}\eta_o(v_n, n, s_c, s_p, c_b, w_s, b, tmp, \eta_t)$ 
if  $\eta_t > 0$  then
   $b \leftarrow \text{reCompute\_noiseband}(b, \eta_t)$ 
end
if Task is terminating then
   $s_c \leftarrow \text{Exit}$ 
  return_state  $\leftarrow Term$  // Changed to terminate task.
end
return return_state

```

---

length window  $w_l$  reacts slower to the recent trends. A stream is in downtrend if the shorter window's result ( $w_s.r$ ) becomes lesser than the longer window's ( $w_l.r$ ). The downtrend behavior is due to any program's tendency to emit relatively smaller IPFVT values during its transitions.

**Definition 5.** Consider two moving windows  $w_s$  and  $w_l$  of a task where  $0 < w_s.l < w_l.l$ , and if  $s_c \in Q$  represents the current locality state of the task, then the corresponding task's memory

locality is in transition iff:

$$(w_s.r < w_l.r) \wedge (s_c = \text{Holding}). \quad (15)$$

The Definition 5 can be abstracted as a function  $T : Q \times \mathcal{F} \times \mathcal{F} \rightarrow \mathbb{Z}^*$  such that:

$$T(s_c, w_s, w_l) = \begin{cases} 3, & (w_s.r < w_l.r) \wedge (s_c = \text{Holding}) \\ 0, & \text{otherwise} \end{cases} \quad (16)$$

### 5.3 Holding Detection

A task can transit to holding state of locality if it emits a value  $v_t \in V_o$  that is also  $m_t \in M$ .

**Definition 6.** Consider a task emits a value  $v_t \in V_o$  at time 0 and  $s_c \in Q$  represent the current state of the task, then  $v_t$  can also be  $m_t \in M$  iff:

$$(v_t > b) \wedge ((s_c = \text{Transition}) \vee (s_c = \text{Init})). \quad (17)$$

The element  $v_t \in V_o$  that caused the system to speculate on a task's holding state is clearly a deviation [8] from normal execution.

Be that as it may, the noise band  $b$  of a task is derived only from the past values, and the future is currently unknown. Hence, concluding prematurely on the deviating  $v_t$  as an element to transit the task to *holding* state might lead to false positives as there might arrive a value in the *near future*  $n$  that is  $v_{t+n} > v_t$ .

By such arguments, the Definition 6 of the holding state can be deemed to be a hypothesis that might transit the task to holding state and also has a possibility to be nullified. To avoid such false conclusions, we first stage the deviated element  $v_t \in V_o$  as a *presumptive* and the algorithm *HypothesisTest* (cf. Algorithm 2) is triggered.

To reduce the probability of false positives, on arrival of a deviated value that satisfies equation 17 it is at first temporarily memorized [30, p. 365] in  $v_p$ . On arrival of every subsequent element  $v_t \in V_o$  after the deviated element is tested for the condition  $v_t > v_p$ . If any of the subsequent elements that arrived after the deviation satisfies the mentioned condition, then  $v_p$  is concluded as  $\eta_o$  (locality noise), else the test continues till the bounded *near future*. If no element satisfies the condition  $v_t > v_p$  for the bounded *near future* then the  $v_p$  is asserted as the *apparent*  $m_t \in M$ .

The distance of the bounded *near future* should be a value that (i) is neither too short that we end up taking a value that should be part of the locality noise, (ii) nor too long that it notices the holding time of the next locality, and (iii) it should also consider the program's nature of exhibiting varying outermost locality degrees during the different instances of program execution. Considering the above mentioned aspects, we define the *near future* length to be  $w_s.l$ , which we surmise to be a reasonable estimation to meet the conditions (i) to (iii).

### 5.4 Online Phase Detector

In this section, the previously discussed sub-algorithms are combined in Algorithm 3 (DetectLocalityChange), which is called on each page fault of a task. The algorithm assumes that windows are initialized and configured by the OS. The variable  $c_g$  is assumed to be managed by the OS. The variable  $s_p$  holds the state of the task in previous page fault (called as immediate previous state). The

**Table 1: Average of the normality test results (P-Value) using various hypothesis tests. All results close to zero confirm the normal distribution.**

| Experiment Name | Shapiro  | D'Agostino | Jarque-Bera |
|-----------------|----------|------------|-------------|
| Mongo DB        | 1.90e-16 | 0.00534    | 0.141       |
| Neo4j           | 6.54e-15 | 2.60e-03   | 0.00194     |
| Memcached       | 0.00000  | 1.04e-30   | 0.00000     |
| Redis           | 1.01e-7  | 2.40e-05   | 3.81e-8     |

function *reCompute\_noiseband* :  $\mathbb{R}^+ \times \mathbb{R}^+ \rightarrow \mathbb{R}^+$  recomputes the average of the noise band  $b$  with the new  $\eta_t \in \eta_o$  element. The algorithm determines a task's state and returns the symbol that caused a transition or returns  $\emptyset$ .

#### Algorithm 4: Detect Outliers on a Complete Sequence $V$

**Input:**  $V$  -The Complete Sequence of IPFVT of a process.  
 $l_o$  -The Length of the sequence  $V$ .

**return value:**  $\hat{M}[]$  -An Array of local Maximas.

**Algorithm** Offline-Eliminate-Outlier():

Initialize the array  $\hat{M}[]$  to 0

Initialize the temporary array  $t[]$  to 0

**while**  $\text{skew}(V) > 0$  and  $l_o! = 0$  **do**

$Std \leftarrow \sigma(V)$

$Mean \leftarrow \mu(V)$

$Limit \leftarrow Mean + (3 * Std)$

**for**  $i \leftarrow 0$ ;  $i < l_o$ ;  $i \leftarrow i + 1$  **do**

**if**  $V[i] > Limit$  **then**

            Append  $V[i]$  to  $t$

**end**

**end**

$l_o \leftarrow$  Negate the length of the  $t$  from  $l_o$

$V \leftarrow$  Remove  $t$  elements from  $V$

    Append  $t$  to  $\hat{M}$

$t[] \leftarrow$  Reinitialize the array to 0

**end**

**return**  $\hat{M}$

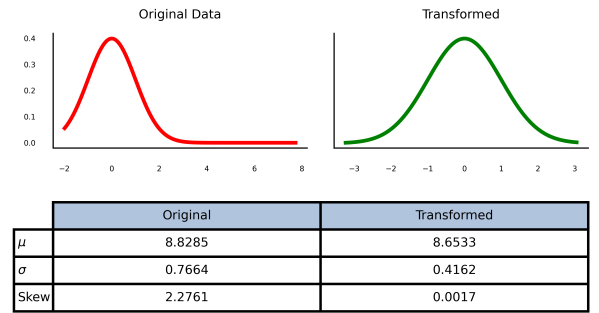


**Figure 3: Transformation of a Positive Leaned Data <sup>1</sup>**

## 6 ESTABLISHING THE GROUND TRUTH

In this section, we design an optimal offline algorithm to assess our approximated online algorithm. To estimate the competitiveness of

<sup>1</sup>The data is derived by running YCSB WorkloadA on MongoDB. Note that the data is logarithmic transformed for easy representation

the proposed online algorithm, we treat the problem as an adversary game [17, p. 3-4] between the proposed online algorithm and an optimal offline algorithm [66, 100]. The optimal offline algorithm is designed and implemented by looking into the complete pre-recorded sequence of the IPFVT. To achieve this, we carried out multiple hypothesis tests [15, 32, 87] using various IPFVT datasets to understand the IPFVT's data distribution. From the empirical results we infer that the IPFVT sequence tends towards normal distribution (some results summarized in Table 1). Generalizing the empirical results using the central limit theorem (CLT) [80, p. 240-243], which establishes that any set of independent random variables when normalized tends towards normal distribution, so is IPFVT.

The local maximas  $M$  in a sequence of IPFVT  $V$  are the data points that diverged from the normal observations. These outliers make the observation heavy-tailed normal distribution [50] and pose a positive leaning skew in the distribution [62, p. 81]. This aspect shows the existence of outliers in the dataset.

A set of elements that needs to be eliminated to attain zero skew value by examining the complete sequence  $V$  in the observation is considered as the *ground truth*, this optimal version of the sequence  $M$  is termed as  $\hat{M}$ . This result is used as a standard reference to assess the correctness of the online algorithm.

Algorithm 4 uses skew correction as the basis for the systematic removal of outliers and uses standard statistical functions – *skew* [46],  $\sigma$ , and  $\mu$  [51]. In worst case, the algorithm will run  $O(n^2)$  and will need  $O(n)$  space, where  $n$  is the number of elements in  $V$ . Figure 3 graphically exemplifies the transformation of one particular instance of  $V$  by the offline algorithm.

## 7 EVALUATION AND EXPERIMENTAL RESULTS

We conducted experiments to answer the following questions: (i) The accuracy of the results generated by our online algorithm compared to the ground truth computed by the optimal offline algorithm over real-world applications (cf. Section 7.4), (ii) The theoretical accuracy of our algorithm and the accuracy of the proposed algorithm compared to other well-known locality transition detection algorithms (cf. Section 7.5), (iii) Empirical overhead of the algorithm on the evaluated system (cf. Section 7.6).

### 7.1 The System Setup

All experiments were performed on an Intel(R) Xeon(R) CPU E3-1545M v5 @ 2.90 GHz, which has 32 KB L1-d cache, 32 KB L1-i cache, 256 KB L2 cache, and 8 MB L3 cache with 16GB of DDR4 main memory @ 2133 Mhz and 32 GB of swap space on SATA-3, 1 TB SSD storage.

The software setup used Linux kernel v4.8.3 and configured to allocate only 4KiB size pages. We used three different software setups for evaluation – the execution system, online evaluation system, and offline evaluation system. The execution system will generate the trace using the workload. The Linux kernel patched with the proposed online algorithm is used for the online evaluation system. The offline evaluation system is a process running on the host system.

### 7.2 Evaluation Model

The result is reduced to a binary classification problem to bring the evaluation to the same scale. Let  $\mathbb{L} = \{0, 1\}$  represent the label space and  $\mathbb{J}$  represent the ordered sequence index with same length and order as the sequence that needs to be evaluated. The binary representation of the data is expressed as a pair  $(t_i, l_i)$  of an ordered sequence where  $t \in \mathbb{J}$  and  $l \in \mathbb{L}$ .

The entire data is denoted as:

$$B = (t_i, l_i), \dots, (t_n, l_n) \subseteq \mathbb{J} \times \mathbb{L} \quad (18)$$

where,

- $t_i \in \mathbb{J}$  is the  $i^{th}$  index.
- $l_i \in \mathbb{L}$  is the label of the  $i^{th}$  index.

We will generate two binary classified sequence  $\mathcal{B}$  and  $\hat{\mathcal{B}}$ . The  $\hat{\mathcal{B}}$  represents the sequence of index values which ground truth identified as phase change points and  $\mathcal{B}$  will represent the sequence of index values which the examined algorithm detected as phase change points, in which:

$$l_i = \begin{cases} 1, & t_i \in K \\ 0, & \text{Otherwise} \end{cases} \quad (19)$$

where  $K$  is a set of indexes that belongs to either  $\mathcal{B}$  or  $\hat{\mathcal{B}}$  depending on the sequence being generated.

The outliers are positive class, and others are seen as negative class. The transformed sequence is used to study two set of measures. The first set of measure is directly derived from the confusion matrix [48] and the second set of measures is derived either from the first set of measurements and represents a higher level of abstraction.

#### 7.2.1 First Level Measurements.<sup>2</sup>

True Positive Rate (TPR) or recall (defined in equation 20) measures the proportion of outlier (true positive) elements correctly identified. In best case the value reaches 1.

$$TPR = \frac{TruePositives}{TruePositives + FalseNegatives} \quad (20)$$

True Negative Rate (TNR) or specificity (defined in equation 21) measures the proportion of normal (true negative) elements correctly identified. In best case the value reaches 1.

$$TNR = \frac{TrueNegatives}{TrueNegatives + FalsePositives} \quad (21)$$

Positive Predictive Value (PPV) or precision (defined in equation 22) measures the proportion of the correct outliers (true positive) results and reaches value 1 on perfect test.

$$PPV = \frac{TruePositives}{TruePositives + FalsePositives} \quad (22)$$

Negative Predictive Value (NPV) measures the proportion of the correctly determined normal element (true negative) results (defined in equation 23) and reaches value 1 if the test result has no false negatives.

$$NPV = \frac{TrueNegatives}{TrueNegatives + FalseNegatives} \quad (23)$$

<sup>2</sup>In the equations TruePositives, FalsePositives, TrueNegatives and FalseNegatives meant the total count of the respective elements.

False Negative Rate (FNR) measures the proportion of wrong normal elements (false negative) results and reaches value 0 on best scenario. The equation is defined in 24.

$$FNR = \frac{FalseNegatives}{TruePositives + FalseNegatives} \quad (24)$$

False Omission Rate (FOR) measures the proportion of normal discoveries that are wrong (false negative) and reaches value 0 on best case. The equation is defined in 25.

$$FOR = \frac{FalseNegatives}{TrueNegatives + FalseNegatives} \quad (25)$$

**7.2.2 Second Level of Measurements.** The classification accuracy is measured using equation 26, where its best case reaches value 1.

$$accuracy(\mathcal{B}, \hat{\mathcal{B}}) = \frac{1}{n} \sum_{i=0}^{n-1} 1(l_i \Leftrightarrow \hat{l}_i) \quad (26)$$

We measured the fraction of the wrong labels (classification loss) using hamming loss and the equation 27 in best case produces value 0 :

$$Loss(\mathcal{B}, \hat{\mathcal{B}}) = \frac{1}{n} \sum_{j=0}^{n-1} 1(l_j \neq \hat{l}_j) \quad (27)$$

We measure  $F_\beta$  score (defined in the equation 28) with two values of  $\beta$ , namely 0.5 and 1.  $F_{0.5}$  gives more weight on precision, as false positives (wrong detection of outliers) are considered expensive while  $F_1$  is a balance on both precision and recall. In both the cases value 1 represents the best case.

$$F_\beta = (1 + \beta^2) \cdot \frac{(precision \cdot recall)}{\beta^2 \cdot precision + recall} \quad (28)$$

### 7.3 DataSet and Workload Design

In the era of Big data and cloud computing, large data sets generated from various sources demand different data models based on the underlying methods used for the data inferencing, dawning to NoSQL. These data models tend to have very complex and cascaded memory access patterns. Based on the underlying data model, we evaluated the four major subcategories of NoSQL databases: Document Store, Graph Store, Key-Value Caching and General Purpose Caching.

We chose the most commonly used open-source databases from each category with their near to reality benchmarking tools [27, 29, 67, 83] (cf. Table 2).

**7.3.1 Trace-Based Evaluation.** To imitate an identical behavior in both online and offline system, the algorithm is assessed using the trace-based evaluation method. An *Execution system* will run forefront the benchmarking tools with workload near to reality and capture the trace sequence of the system.

**Definition 7.** Each trace element  $T_{idx}$  in a trace sequence  $T$  is a 3-tuple  $T_{idx} = (idx, pid, v)$ , emitted on every page fault. The  $idx \in \mathbb{W}$  represents the page fault's sequence number,  $pid \in \mathbb{N}$  is the process id of the corresponding task that generated the page fault, and  $v \in \mathbb{R}_0^+$  is the corresponding IPFVT.

The trace will be used to run the online and offline algorithms. To facilitate the approach, we designed a new tool named MemGobbler

**Table 2: DB Types and Corresponding Benchmark Tools.**

| Type                    | DataBase  | BenchMark Tool    |
|-------------------------|-----------|-------------------|
| General Purpose Caching | Memcached | Memtier           |
| Document Store          | MongoDB   | YCSB              |
| Graph Store             | Neo4j     | GraphDB-BenchMark |
| Key-Value Based Caching | Redis     | Memtier           |

to ease the evaluation of the online algorithm. The tool will run in the userspace and generate a page fault at the right interval such that it exacts the given trace sequence  $V$ .

**Table 3: Memcached Experiment Workload**

| Display Name | Get:Set Ratio        | Value Size |
|--------------|----------------------|------------|
| Mc-Set1      | 50 : 50              | 270B       |
| Mc-Set2      | 1 : 30               | 270B       |
| Mc-Set3      | 50 : 50              | 750B       |
| Mc-Set4      | 1 : 30               | 750B       |
| Mc-Set5      | 50 : 50              | 1000B      |
| Mc-Set6      | 1 : 30               | 1000B      |
| Mc-Set7      | 50 : 50              | 1MB        |
| Mc-Set8      | 1 : 30               | 1MB        |
| Mc-Set9      | 1 : 30 (50 Threads)  | 1MB        |
| Mc-Set10     | 1 : 30 (100 Threads) | 1MB        |

**Table 4: MongoDB Experiment Workload**

| Display Name | Experiment Name                     | Record Count |
|--------------|-------------------------------------|--------------|
| MngDB-Set1   | 50% Read and 50% Write              | 500000       |
| MngDB-Set2   | 95% Read and 5% Write               | 500000       |
| MngDB-Set3   | 95% paging scan                     | 500000       |
| MngDB-Set4   | 50% Read and 50% Write              | 1000000      |
| MngDB-Set5   | 95% Read and 5% Write               | 1000000      |
| MngDB-Set6   | 95% paging scan                     | 1000000      |
| MngDB-Set7   | 50% Read, 50% Write and 50 Threads  | 500000       |
| MngDB-Set8   | 50% Read, 50% Write and 100 Threads | 500000       |

**Table 5: Neo4j Experiment Workload**

| Display Name | Experiment Name | Node Count |
|--------------|-----------------|------------|
| Neo4j-Set1   | Aggregation     | 10000      |
| Neo4j-Set2   | HardPath        | 10000      |
| Neo4j-Set3   | Neighbour       | 100000     |
| Neo4j-Set4   | Neighbour2      | 100000     |
| Neo4j-Set5   | Shortest Path   | 100000     |

**7.3.2 Workload Design.** In MongoDB [2], we performed update heavy, read-heavy, and read the latest operations (summarized in the Table 4), in which Zipfian distribution [20] is used for record selection to imitate real world scenario. Zipfian distribution imitates the real-world scenario by gradually decaying the hotness of the previous popular records. To determine the accuracy in multi-threaded environment the last two results in the Table 4 are run with 50 and 100 parallel thread requests.

**Table 6: Redis Experiment Workload**

| Display Name | Get:Set Ratio        | Value Size |
|--------------|----------------------|------------|
| Red-Set1     | 50 : 50              | 270B       |
| Red-Set2     | 1 : 30               | 270B       |
| Red-Set3     | 50 : 50              | 750B       |
| Red-Set4     | 1 : 30               | 750B       |
| Red-Set5     | 50 : 50              | 1000B      |
| Red-Set6     | 1 : 30               | 1000B      |
| Red-Set7     | 50 : 50              | 1MB        |
| Red-Set8     | 1 : 30               | 1MB        |
| Red-Set9     | 1 : 30 (50 Threads)  | 1MB        |
| Red-Set10    | 1 : 30 (100 Threads) | 1MB        |

**Table 7: Consolidated First Level Results.**

| DB Name              | TPR         | TNR        | PPV        | NPV         | FNR         | FOR         |
|----------------------|-------------|------------|------------|-------------|-------------|-------------|
| Memcached            | 0.95        | 1.0        | 1.0        | 0.95        | 0.04        | 0.04        |
| MongoDB              | 0.84        | 1.0        | 1.0        | 0.86        | 0.15        | 0.13        |
| Neo4j                | 0.87        | 1.0        | 1.0        | 0.89        | 0.12        | 0.10        |
| Redis                | 0.92        | 1.0        | 1.0        | 0.93        | 0.07        | 0.06        |
| <b>Grand Average</b> | <b>0.89</b> | <b>1.0</b> | <b>1.0</b> | <b>0.91</b> | <b>0.10</b> | <b>0.08</b> |

We evaluated caching databases: Memcached [1] and Redis [25] using real world inspired workloads. The maximum number of key-value elements is restricted to 5 GB of the main memory. The accuracy is also tested on multi-threaded environment with up to 100 parallel thread requests (workload summary in Table 3 and Table 6).

Neo4j [3] graph-based database is evaluated using Pokec online social network in SNAP datasets [69] as workload and made to run the memory-intensive operations detailed in Table 5.

## 7.4 Result Summary and Analysis

Figure 4 and the table 7 show the first level results. Due to cautious decision-making nature of the online algorithm in deciding the outlier, we notice no false positives (Type 1 error) for all the experiments; hence zero false discovery rate (FDR) and false positive rate (FPR). The average TPR (sensitivity) of the algorithm which decides the proportion of outliers correctly detected is 89% and the specificity (TNR) is 100%. FNR (miss rate) and FOR which determines the portion of wrongly determined values is on an average 10% and 8%, respectively.

Figure 5 and the table 8 show the second-level experiment results. The average accuracy of the experiment is 89% and, in best cases it reached 100%. The  $F_{0.5}$  score and  $F_1$  score is on an average 74% and 65% respectively. In specific experiments, the  $F_{0.5}$  score and  $F_1$  score was 100%. The algorithm's loss or inaccuracy is on an average 10 percent.

We notice in an extreme multithreaded environment an average accuracy reduced by 2% (from 91% to 89%),  $F_1$  score reduced by 4% (from 69% to 65%),  $F_{0.5}$  score reduced by 3% (from 77% to 74%) and loss increased by 2% (from 8% to 10%). This loss of accuracy in Linux is due to non existence of per thread memory management data structure. We see this is inconsequential due to non existence of false positives, i.e., the algorithm may have missed to identify certain phases, but never identified any phase wrongly. Though

**Table 8: Consolidated Second Level Results.**

| DB Name              | Accuracy    | F0.5-Score  | F1-Score    | Loss        |
|----------------------|-------------|-------------|-------------|-------------|
| Memcached            | 0.95        | 0.74        | 0.69        | 0.04        |
| MongoDB              | 0.84        | 0.67        | 0.59        | 0.15        |
| Neo4j                | 0.87        | 0.73        | 0.66        | 0.12        |
| Redis                | 0.90        | 0.78        | 0.63        | 0.09        |
| <b>Grand Average</b> | <b>0.89</b> | <b>0.74</b> | <b>0.65</b> | <b>0.10</b> |

hyper-parameters have clear semantics, in order to have no preparatory dependence, the experiment is performed using golden ratio values  $w_s.l = 7$  and  $w_l.l = 13$ .

## 7.5 Theoretical and Baseline Evaluation

To understand the theoretical accuracy of the proposed algorithm the evaluation is done using memory-pattern based synthetic trace generation methodology. This evaluation is done to understand (i) theoretical accuracy of the proposed algorithm, (ii) the effectiveness of our evaluation technique which we used over real world applications, and (iii) the accuracy of the proposed algorithm compared to other locality change detection algorithms.

The synthetic trace generation approach is an oblivious adversary model [17, p. 45] which depicts most user programs, i.e., it generates the complete synthetic trace sequence in advance and unaware of the behavior of the potential users. The trace sequence is generated by strictly following the phase-transition behavior [40, 49, 53, 65]. The methodology will generate a reference string sequence (cf. Section 2) from which we derive different locality measuring models and used them to evaluate other existing locality detection algorithms. This approach enabled us to perform a hand-on-hand comparison of the proposed algorithms against other well-known locality detection algorithms.

**Definition 8.** Each trace element  $T_{idx}^s$  of the synthetic trace  $T^s$  is a 5-tuple  $T_{idx}^s = (idx, r, rud, i, m)$ , where  $idx \in \mathbb{W}$  is the sequence index of the trace,  $r \in \mathbb{W}$  is the page number referenced at that instance,  $rud \in \mathbb{W}$  is the reuse distance of that instance,  $i \in \mathbb{R}_0^+$  is the IPFVT of that instance that is set to 0 if the corresponding instance does not hold a valid IPFVT, and  $m \in \{0, 1\}$  is the phase marker set to 1 if the instance is beginning of the phase else is set to 0.

Two extreme memory access patterns are used to generate the reference string namely sequential and random [75]. We notice that these two extreme memory access patterns are sufficient to cover all the other intermediate access patterns for the used measuring metrics. We executed various workloads on each memory access pattern based on memory size ranging from 1MB to 130MB. Each evaluation is a 5MB incremental step, i.e., each access pattern assessed with 26 experiments, so 52 experiment runs were conducted in total.

The baseline evaluation is made using four well-known algorithms (cf. Table 9), which we observe to cover most ideas regarding phase change detection. The first four algorithms in the Table 9 are offline that will need the complete trace to decide on the phase change, while the last two are online algorithms. The algorithms named Proposed Offline and Proposed Online represent our algorithms mentioned in the Section 6 and Section 5 respectively.

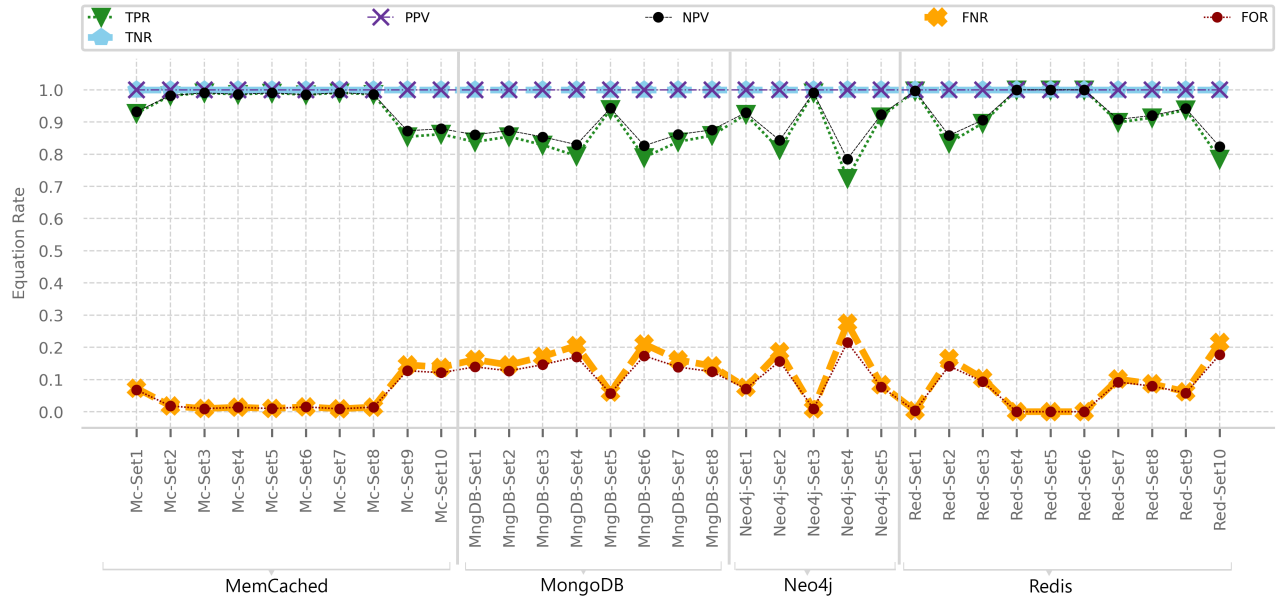


Figure 4: First Level Results. The horizontal axis represents the Display Name in the corresponding experiment table.

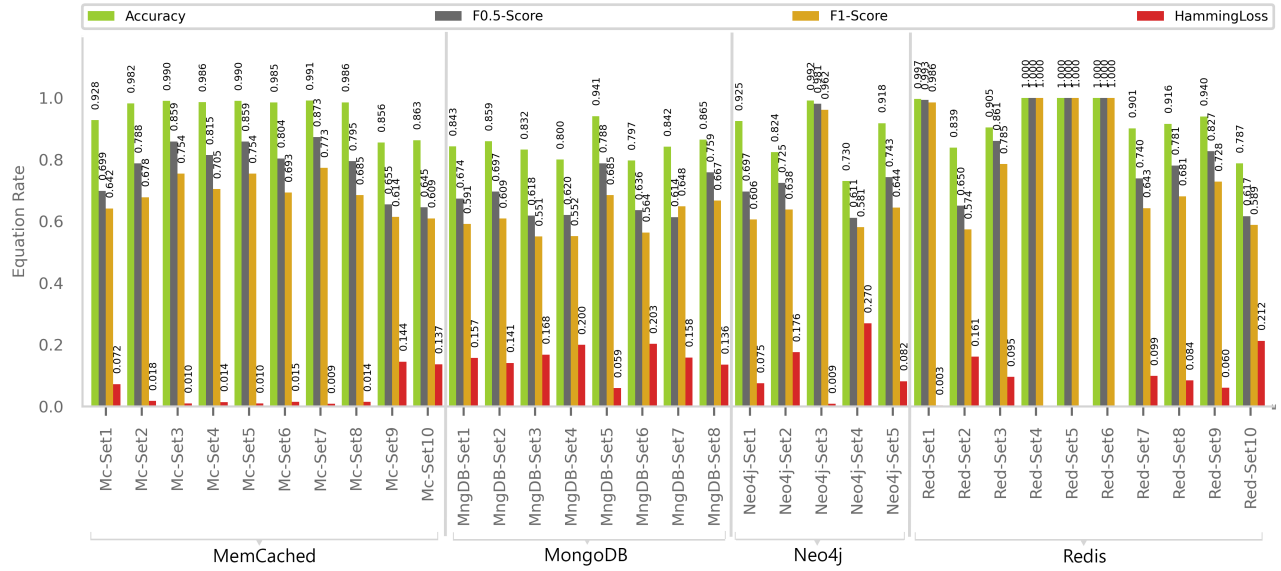


Figure 5: Second Level Results. The horizontal axis represents the Display Name in the corresponding experiment table.

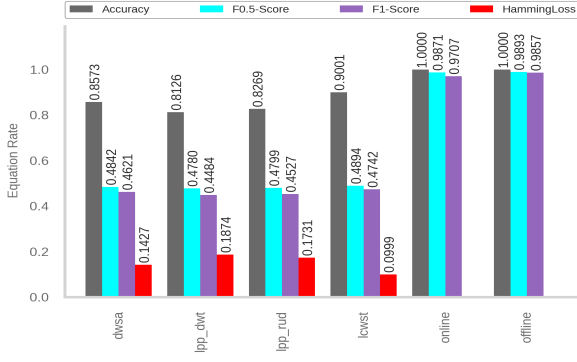
Dhodapkar et al. [44] uses reference pattern and fixed interval sampling, while Shen et al. [88] and Ding et al. [45] uses reuse distance [109]. Zhao et al. [107] uses variance in the Translation Lookaside Buffer (TLB) to determine the locality change.

We transformed the results into binary sequence to compare the marked synthetic trace with the result generated by the algorithms, where the phase change points are positive class, and others are negative class. The binary sequence generated using the marked

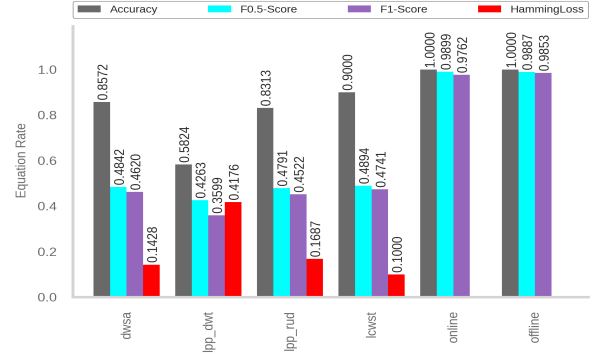
synthetic trace is considered the ground truth. The transformed result is evaluated using classification assessment metrics described in 7.2.2. The results are summarized in Figures 6a and 6b.

We notice that the proposed online algorithm has a theoretical accuracy of 100%, and F0.5 and F1 scores are around 98% with zero loss. We see very similar results with the optimal offline algorithm.

The result which compares the proposed ground truth computation (cf. Section 6) with the theoretically phase-marked traces,



(a) Random Memory Pattern



(b) Sequential Memory Pattern

Figure 6: Summarized theoretical evaluation of the experiments. Labels correspond to Display Name in Table 9.

Table 9: Baseline Comparison of the Algorithms. The column *space* and *time* represents the corresponding axiomatic complexity of the algorithm.

| Algorithm             | Display Name | Space  | Time     | HW Support |
|-----------------------|--------------|--------|----------|------------|
| Dhodapkar et al. [44] | dwsa         | $O(n)$ | $O(n^2)$ | Yes        |
| Shen et al. [88]      | lpp_dwt      | $O(n)$ | $O(n)$   | Yes        |
| Ding et.al [45]       | lpp_rud      | $O(n)$ | $O(n)$   | Yes        |
| Proposed Offline      | offline      | $O(n)$ | $O(n^2)$ | No         |
| Zhao et.al [107]      | lcwst        | $O(1)$ | $O(1)$   | Yes        |
| Proposed Online       | online       | $O(1)$ | $O(1)$   | No         |

demonstrates that the proposed optimal offline algorithm is a reliable method to compute the ground truth for phase transition detection. Further, the evaluation approach we applied over the real world applications is a trivial, trustworthy, and can be used as a de facto approach to mark the phases.

Our proposed approach, which uses IPFVT as metric for outlier detection enabled this accuracy. Further, the concise representation of IPFVT allowed our offline algorithm to generate the required result with less than 1% of computation compared to other offline algorithms that is based on reference string or reuse distance as the measuring metric.

Other phase-transition detection algorithms had an accuracy ranging between 57 and 90 percent, however, demonstrated very low F0.5 and F1 scores ranging between 36 and 49 percent, which are considered important metrics in determining the right accuracy balance between precision and recall in imbalanced classification problems.

## 7.6 Empirical Overhead Analysis

The proposed implementation induced two additional logic to the Linux kernel: (i) additional computation induced to the Scheduling Decision function (SD) to compute the  $S_p$  (scheduling noise), which will be removed from IPFVT, (ii) the computation induced to the Page Fault (PF) Interrupt Service Routine (ISR) to compute the IPFVT (cf. Section 4.1) along with the DFA transition logic, which is triggered on each arrival of an IPFVT to decide the locality

transition. If  $SD_k^t$  is the time taken to compute one SD without our

Figure 7: The computation overhead between page faults. There could be  $k$  scheduling decisions between two page faults, where  $k \geq 0$ .

patch at  $k$ 'th instance and  $PF_n^t$  is the time taken to compute one PF without our patch at the  $n$ 'th instance then, we name the additional time induced by our logic to the  $SD_k^t$  as  $AL\_SD_k$  and the additional time induced by our logic to the  $PF_n^t$  as  $AL\_PF_n$ . Figure 7 describes an abstract relation of the overall computation. If  $OC_k^{sd}$  is the time taken by original SD computation between two page faults, which is defined as:

$$OC_k^{sd} = \begin{cases} \sum_{i=1}^k SD_k^t, & k > 0 \\ 0, & \text{otherwise} \end{cases} \quad (29)$$

Then, the time taken by one original computation  $OC_n$  at the instance  $n$  is calculated using:

$$OC_n = OC_k^{sd} + PF_n^t \quad (30)$$

If  $AT_k^{sd}$  is the additional time introduced into SD computation between two faults, which is defined as:

$$AT_k^{sd} = \begin{cases} \sum_{i=1}^k AL\_SD_k, & k > 0 \\ 0, & \text{otherwise} \end{cases} \quad (31)$$

Then, additional Time  $AT_n$  induced by our proposed algorithm for the instance  $n$  is calculated using:

$$AT_n = (AT_k^{sd} + AL\_PF_n) - C \quad (32)$$

where,  $C$  is the approximated constant overhead induced by the overhead computation framework. The overhead percent  $OH_n$  induced by our implementation ( $AT_n$ ) in addition to the original computation ( $OC_n$ ) for the instance  $n$  is:

$$OH_n = \frac{AT_n}{OC_n} \cdot 100 \quad (33)$$

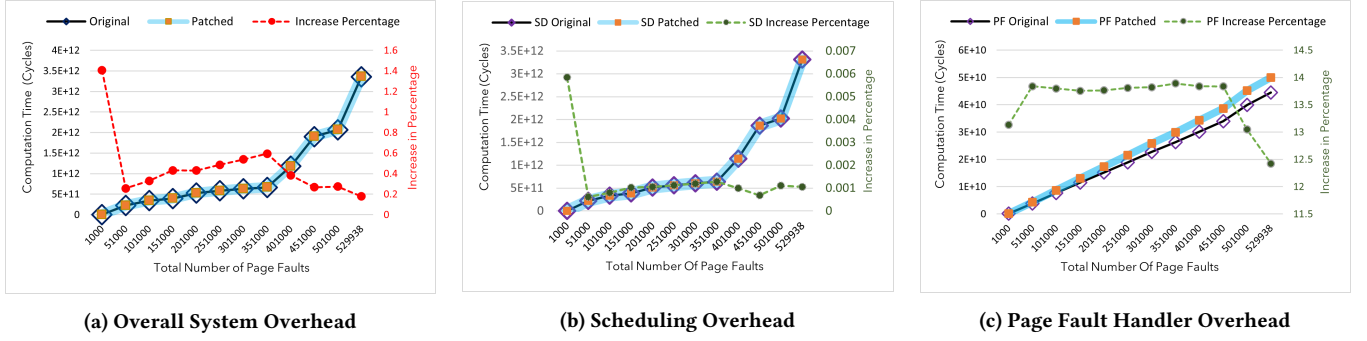


**Figure 8: Overhead induced by the online algorithm. The primary axis curves lie over each other due to very low overhead.**

The primary y-axis in Figure 8a, Figure 8b and Figure 8c represents the computation time in cycles. The Original and Patched represent the computation time with and without our implementation respectively. The secondary y-axis in the figures represents the induced overhead in percentage. The Figure 8b and Figure 8c represent the overhead introduced into the scheduler and page fault ISR respectively.

We analyzed the overhead caused by the algorithm with up to 529974 page faults. The analysis shows that the overall algorithm induces a negligible overhead ranging [0.02, 1.3] percent and averaging around 0.506 percent (cf. Figure 8a). The negligible overhead observed (1.4% in worst case) makes the algorithm adaptable even to latency-sensitive and real-time systems. Our solution added around 7378 cycles (averaging 0.004%) per scheduling decision which average around  $9.046E8$  cycles (cf. Figure 8b). Further, our implementation added around 11240 cycles (averaging 13.40%) to the original page fault interrupt service routine computation, which average around 83872 cycles per page fault (cf. Figure 8c).

## 8 RELATED WORK

The scientific research on detecting the locality changes using phase transition techniques has been crucial in adapting multi-configuration dynamic systems according to the program behavior to achieve higher energy efficiency and performance.

In hardware, the techniques are incorporated to dynamically configure the line size or total size of the TLB or cache, or dynamically control the depth of the branch predictors, or dynamically close down certain sections of the issue window.

The authors of [11] implemented a dynamically size configurable L1 cache, using fixed-interval-based phase change determination. The variance in specific hardware counters like conditional branch miss count between two successful intervals determines the phase state. The determined phase state, along with the current cache configuration, determines the subsequent configuration of the cache.

In [92] the authors use a hardware-implemented working set signature comparison method for dynamic microarchitecture configuration of the CPU to alter the length of the branch predictor, determine the enablement of certain sections of the pipeline, and many more.

Hassan et al. [57] predicts the locality change for dynamic workload characterisation by building hardware based cumulative distribution function of the reuse distance, based on the Least Recently Used Stack Model (LRUSM) on the precollected reference string and uses markov model to predict the transition.

On the software side, the technique is used for compiler based dynamic code optimization. The software oriented methods uses techniques based on basic block vectors and working set signatures.

The authors of [89–91] propose using a hardware-independent metric named basic block vectors (BBVs) to detect the phase changes offline to determine the simulation points, used to understand a processor’s cycle-level behavior. Phase changes are detected when the Manhattan distance between consecutive BBVs exceeds a preset threshold.

Pusukuri et al. [84] predicts the locality phase change to determine working set size using TLB (Translation Lookaside Buffer) miss rate and LLC (Last Level Cache) miss rate as predictor variables along with the number of pages referenced by a process at a given instance of time to train three supervised learning models to select the better among the models for prediction.

In [60, 61] the authors propose a positional adaptation technique at subroutine level granularity to identify the phases. The positional approach uses the similarity between different invocations of the same code section to identify the phases. The framework is used to manage energy and temperature dynamically, and combines an offline approach with the hardware unit to make the decisions.

Zhao et al. [107] uses locality change detection in scalable resource systems using an approach called Intermittent Memory Tracking (IMT). The IMT detects the locality phase change behavior by observing and detecting the sudden change in PMC (Performance Monitoring Counter) events of the DTLB (Data Translation Lookaside Buffer) miss. The algorithm does not consider the scenario where the WSS (Working Set Size) of a process might be larger than the TLB (Translation Lookaside Buffer) and might add context switch overhead.

In most of the research papers mentioned above, we notice that the actual accuracy of the phase-detection algorithms is not evaluated; instead, just the applied notion of the results is presented.

The Miss Ratio Curve (MRC) [103] is among the primary tool for cache performance estimation uses locality principle as basis. The majority of the research tries to reduce the taxonomy in deriving the MRC using approaches like stack distance [6, 14, 76, 97, 110]

and reuse time [13, 16, 59, 102]. The MRC based approaches are based on reference string and best traced with hardware support.

The authors of [108, 111] derive the MRC using a stack of hot page reference traces using software-only approach by toggling the read-write permission, which induces very high context-switch overhead [19] impeding the usage on real-world workload.

The rest of this section summarizes some of the current prevailing issues in the operating system and different attempts to address them, which we perceive will have a better solution using our proposed algorithm.

The observations of memory locality behavior had conventionally approximated into online reclamation algorithms like Working Set Clock (WSCLOCK) [23], Page Fault Frequency (PFF) [26], 2Q [64]. A set of hyperparameters and the scan frequency (scan rate policy: determines the intervals at which the reclamation algorithm should run) determine the accuracy of these algorithms, which can be automated using the proposed phase-transition algorithm.

Many authors [81, 82] address the Linux page reclamation Jitter by providing methods to tune watermarks. The proposed approaches statically determine the watermark and require error-prone manual admin intervention to set the hyperparameter. This can be automated using our proposed approach resulting in dynamic watermarks.

Predictive memory reclamation and compaction [10] attempts to reduce the memory stall due to direct reclamation, uses curve-fitting approach to analyze the page allocation trend to decide whether the kswapd daemon needs to sleep or continue the reclamation.

Linux idle memory tracking [34–37], which is used to estimate the working set size (WSS) of a process [54, 55] using fixed interval polling, might work more accurately if run at right meaningful intervals, i.e., the moments a locality shift happens.

Workaround engineering, like earlyoom [96], nohang [9] tries to prevent the scalable high performance system from entering the unresponsive state with the cost of running ten times per second to check the available memory. These approaches have more chances to terminate the processes whose pages are yet to be tested and add considerable overhead to the system due to context switches.

To the best of our knowledge, the presented pure software solution, which detects the working set change at execution time using phase transition model with no preparatory dependence has not been explored in prior works.

## 9 CONCLUSION AND FUTURE WORK

We proposed a novel software-based solution for embedding knowledge on programs' memory allocation behavior into an operating system's memory management software. We used a novel and reliable binary classification based evaluation technique, conducted 33 experiments on four distinct databases with varied workloads and evaluated nine varied metrics. The assessment showed that despite no preparation and prior knowledge of the executing task, the proposed solution achieved 89% accuracy on real world applications. The sensitivity and specificity of the solution were 89% and 100%. The non-existence of false positives in any experiments shows that no normal value was wrongly identified as an outlier. Further, we conducted theoretical evaluation to perform baseline comparison with already existing locality change detection algorithms, and also

used the same to value the efficiency of our evaluation technique we used on real-world applications. The theoretical accuracy of our technique is 100% with F1-Score and F0.5-Score averaged around 98% with zero loss. Finally, we showed that the empirical overhead of the algorithm is on an average 0.506% making the algorithm suitable for real-time systems.

The proposed per-process online phase detection algorithm that executes in constant time and space complexity opens several opportunities that we plan to explore in the future. The implementation of the algorithm as a DFA that detects the essential moments in which a task switches to a new working set can provide meaningful sporadic trigger events to an operating system's memory management decision-making process. Furthermore, we intend to apply the proposed algorithm to reformulate the scanning process in the reclamation algorithm to maintain only working set pages in the DRAM-based main memory while others pages are evicted from it and secured in non-volatile memory or storage. This will be of great value in scalable systems to achieve higher performance and energy efficiency.

## ACKNOWLEDGMENTS

The authors thank Yugeshwar Ponnappillai for clarifying the fundamentals of statistics and data science. The authors thank Kira Kraft for proofreading the mathematical formalization presented in this paper. The authors thank Prof. Dr. Ralf G. Herrtwich and K R Baalaji of Nvidia for being supportive in making of this paper. This work was supported by the Fraunhofer High Performance Center for Simulation- and Software-based Innovation.

## REFERENCES

- [1] 2021. Memcached Official Website. <http://memcached.org>. (Visited on 2021-May-24).
- [2] 2021. MongoDB Official Website. <https://www.mongodb.com/>. (Visited on 2021-May-24).
- [3] 2021. Neo4j Official Website. <https://neo4j.com/>. (Visited on 2021-May-24).
- [4] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. 1971. Principles of Optimal Page Replacement. *J. ACM* 18, 1 (Jan. 1971), 80–93. <https://doi.org/10.1145/321623.321632>
- [5] Susanne Albers, Lene M. Favrholdt, and Oliver Giel. 2002. On Paging with Locality of Reference. In *Proceedings of the Thirty-fourth Annual ACM Symposium on Theory of Computing* (Montreal, Quebec, Canada) (STOC '02). ACM, New York, NY, USA, 258–267. <https://doi.org/10.1145/509907.509949>
- [6] George Almási, Călin Cașcaval, and David A. Padua. 2002. Calculating Stack Distances Efficiently. In *Proceedings of the 2002 Workshop on Memory System Performance* (Berlin, Germany) (MSP '02). ACM, New York, NY, USA, 37–43. <https://doi.org/10.1145/773146.773043>
- [7] Advanced Micro Devices Inc (AMD). 2018. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*.
- [8] Andreas Arning, R. Agrawal, and P. Raghavan. 1996. A Linear Method for Deviation Detection in Large Databases. In *KDD*.
- [9] Alexey Avramov. 2019. nohang. <https://github.com/KAMiKAZOW/nohang>. (Visited on 2021-May-24).
- [10] Khalid Aziz. 2019. Add predictive memory reclamation and compaction. <https://lwn.net/Articles/796083/> (Visited on 2021-May-24).
- [11] R. Balasubramanian, D. Albones, A. Buyuktosunoglu, and S. Dwarkadas. 2000. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*. 245–257. <https://doi.org/10.1109/MICRO.2000.898075>
- [12] A. P. Batson and A. W. Madison. 1976. Measurements of Major Locality Phases in Symbolic Reference Strings. In *Proceedings of the 1976 ACM SIGMETRICS Conference on Computer Performance Modeling Measurement and Evaluation* (Cambridge, Massachusetts, USA) (SIGMETRICS '76). ACM, New York, NY, USA, 75–84. <https://doi.org/10.1145/800200.806184>
- [13] N. Beckmann and D. Sanchez. 2016. Modeling cache performance beyond LRU. In *2016 IEEE International Symposium on High Performance Computer Architecture*

- (HPCA). 225–236. <https://doi.org/10.1109/HPCA.2016.7446067>
- [14] B. T. Bennett and V. J. Kruskal. 1975. LRU Stack Processing. *IBM J. Res. Dev.* 19, 4 (July 1975), 353–357. <https://doi.org/10.1147/rd.194.0353>
  - [15] Anil K. Bera and Carlos M. Jarque. 1981. Efficient tests for normality, homoscedasticity and serial independence of regression residuals: Monte Carlo Evidence. *Economics Letters* 7, 4 (1981), 313–318. [https://doi.org/10.1016/0165-1765\(81\)90035-5](https://doi.org/10.1016/0165-1765(81)90035-5)
  - [16] E. Berg and E. Hagersten. 2004. StatCache: a probabilistic approach to efficient and accurate data locality analysis. In *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software*, 2004. 20–27. <https://doi.org/10.1109/ISPASS.2004.1291352>
  - [17] Allan Borodin and Ran El-Yaniv. 1998. *Online computation and competitive analysis*. Cambridge University Press. I–XVIII, 1–414 pages.
  - [18] Daniel Bovet and Marco Cesati. 2005. *Understanding The Linux Kernel*. O'Reilly & Associates Inc.
  - [19] Björn B. Brandenburg. 2011. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. Ph.D. Dissertation. Chapel Hill, NC, USA. Advisor(s) Anderson, James H. AAI3502550.
  - [20] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. 1999. Web caching and Zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, Vol. 1. 126–134 vol.1. <https://doi.org/10.1109/INFCOM.1999.749260>
  - [21] Shakeel Butt. 2019. Proactively reclaiming idle memory. (2019). <https://lwn.net/Articles/787611/> (Visited on 2021-May-24).
  - [22] Giorgio Buttazzo, Giuseppe Lipari, L. Abeni, and Marco Caccamo. 2005. Soft real-time systems: Predictability vs. efficiency. (01 2005). <https://doi.org/10.1007/0-387-28147-9>
  - [23] Richard W. Carr and John L. Hennessy. 1981. WSCLOCK&mdash;a Simple and Effective Algorithm for Virtual Memory Management. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) (SOSP '81). ACM, New York, NY, USA, 87–95. <https://doi.org/10.1145/800216.806596>
  - [24] Samuel T. Chanson and Bernard Law. 1981. *Strategy-Independent Program Restructuring Based on Bounded Locality Intervals*. Technical Report. Vancouver, BC, Canada, Canada.
  - [25] S. Chen, X. Tang, H. Wang, H. Zhao, and M. Guo. 2016. Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis. In *2016 IEEE Trustcom/BigDataSE/ISPA*. 1660–1667. <https://doi.org/10.1109/TrustCom.2016.0255>
  - [26] Wesley W. Chu and Holger Opderbeck. 1972. The Page Fault Frequency Replacement Algorithm. In *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I (Anaheim, California) (AFIPS '72 (Fall, part I))*. ACM, New York, NY, USA, 597–609. <https://doi.org/10.1145/1479992.1480077>
  - [27] Marek Ciglan, Alex Averbuch, and Ladislav Hluchy. 2012. Benchmarking Traversal Operations over Graph Databases. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops (ICDEW '12)*. IEEE Computer Society, Washington, DC, USA, 186–189. <https://doi.org/10.1109/ICDEW.2012.47>
  - [28] Edward G. Coffman, Jr. and Peter J. Denning. 1973. *Operating Systems Theory*. Prentice Hall Professional Technical Reference.
  - [29] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) (SoCC '10). ACM, New York, NY, USA, 143–154. <https://doi.org/10.1145/1807128.1807152>
  - [30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.
  - [31] Intel Corporation. 2007. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*. Intel Corporation.
  - [32] R. D'Agostino and A. Belanger. 1990. A Suggestion for Using Powerful and Informative Tests of Normality. *The American Statistician* 44 (1990), 316–321.
  - [33] Beibei Yang David Dietrich, Barry Heller. 2015. *Data Science and Big Data Analytics: Discovering, Analyzing, Visualizing and Presenting Data* (1st ed.). Wiley Publishing.
  - [34] Vladimir Davydov. 2015. Idle Memory Tracking Article. <https://lwn.net/Articles/643578/>. (Visited on 2021-May-24).
  - [35] Vladimir Davydov. 2015. Idle Memory Tracking Patch-V1. <https://lkml.org/lkml/2015/4/28/224/>. (Visited on 2021-May-24).
  - [36] Vladimir Davydov. 2015. Idle Memory Tracking Patch-V2. <https://lkml.org/lkml/2015/4/7/260>. (Visited on 2021-May-24).
  - [37] Vladimir Davydov. 2015. Idle Memory Tracking Patch-V3. <https://lkml.org/lkml/2015/3/18/794>. (Visited on 2021-May-24).
  - [38] Peter J. Denning. 1968. The Working Set Model for Program Behavior. *Commun. ACM* 11, 5 (May 1968), 323–333. <https://doi.org/10.1145/363095.363141>
  - [39] Peter J. Denning. 1972. On Modeling Program Behavior. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference* (Atlantic City, New Jersey) (AFIPS '72 (Spring)). ACM, New York, NY, USA, 937–944. <https://doi.org/10.1145/1478873.1478998>
  - [40] P. J. Denning. 1980. Working Sets Past and Present. *IEEE Trans. Softw. Eng.* 6, 1 (Jan. 1980), 64–84. <https://doi.org/10.1109/TSE.1980.230464>
  - [41] Peter J. Denning. 2005. The Locality Principle. *Commun. ACM* 48, 7 (July 2005), 19–24. <https://doi.org/10.1145/1070838.1070856>
  - [42] Peter J. Denning and Kevin C. Kahn. 1975. A Study of Program Locality and Lifetime Functions. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles* (Austin, Texas, USA) (SOSP '75). ACM, New York, NY, USA, 207–216. <https://doi.org/10.1145/800213.806539>
  - [43] Peter J. Denning and Kevin C. Kahn. 1976. An L=S Criterion for Optimal Multiprogramming. In *Proceedings of the 1976 ACM SIGMETRICS Conference on Computer Performance Modeling Measurement and Evaluation* (Cambridge, Massachusetts, USA) (SIGMETRICS '76). ACM, New York, NY, USA, 219–229. <https://doi.org/10.1145/800200.806198>
  - [44] A.S. Dhodapkar and J.E. Smith. 2002. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings 29th Annual International Symposium on Computer Architecture*. 233–244. <https://doi.org/10.1109/ISCA.2002.1003581>
  - [45] Chen Ding and Yutao Zhong. 2003. Predicting Whole-Program Locality through Reuse Distance Analysis. *SIGPLAN Not.* 38, 5 (May 2003), 245–257. <https://doi.org/10.1145/780822.781159>
  - [46] David P. Doane and Lori E. Seward. 2011. Measuring Skewness: A Forgotten Statistic? *Journal of Statistics Education* 19, 2 (2011), null. <https://doi.org/10.1080/10691898.2011.11889611> arXiv:https://doi.org/10.1080/10691898.2011.11889611
  - [47] Ronald Fagin and Thomas G. Price. 1978. Efficient Calculation of Expected Miss Ratios in the Independent Reference Model. *SIAM J. Comput.* 7, 3 (1978), 288–297.
  - [48] Tom Fawcett. 2006. An Introduction to ROC Analysis. 27, 8 (June 2006), 861–874. <https://doi.org/10.1016/j.patrec.2005.10.010>
  - [49] Domenico Ferrari. 1974. Improving Locality by Critical Working Sets. *Commun. ACM* 17, 11 (Nov. 1974), 614–620. <https://doi.org/10.1145/361179.361195>
  - [50] SERGEY FOSS, DMITRY KORSHUNOV, and STAN ZACHARY. 2009. An Introduction to Heavy-tailed and Subexponential Distributions.
  - [51] S. Ghahramani. 2015. *Fundamentals of Probability: with Stochastic Processes, Third Edition*. CRC Press. <https://books.google.de/books?id=mmrdCgAAQBAJ>
  - [52] Mel Gorman. 2004. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
  - [53] Gordon Scott Graham. 1976. *A Study of Program and Memory Policy Behaviour*. Ph.D. Dissertation. West Lafayette, IN, USA. AAI7715411.
  - [54] Brenden Gregg. 2021. Working Set Tracking Using Idle Memory Tracking-Article. <http://www.brendangregg.com/wss.html>. (Visited on 2021-May-24).
  - [55] Brenden Gregg. 2021. Working Set Tracking Using Idle Memory Tracking-Patch. <https://github.com/brendangregg/wss>. (Visited on 2021-May-24).
  - [56] Dan Gusfield. 1997. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press.
  - [57] Rahman Hassan and Antony Harris. 2007. A Hybrid Markov Model for Accurate Memory Reference Generation. *Lecture Notes in Engineering and Computer Science* (01 2007), 550–555.
  - [58] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
  - [59] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2016. Kinetic Modeling of Data Eviction in Cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 351–364. <https://www.usenix.org/conference/atc16/technical-sessions/presentation/hu>
  - [60] Michael Huang, Jose Renau, Seung-Moon Yoo, and Josep Torrellas. 2000. A Framework for Dynamic Energy Efficiency and Temperature Management. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture* (Monterey, California, USA) (MICRO 33). Association for Computing Machinery, New York, NY, USA, 202–213. <https://doi.org/10.1145/360128.360149>
  - [61] M. C. Huang, J. Renau, and J. Torrellas. 2003. Positional adaptation of processors: application to energy reduction. *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* (2003), 157–168.
  - [62] B. Illowsky and S. Dean. 2008. *Collaborative Statistics*. Illowsky Publishing. <https://books.google.de/books?id=p2RtPQAACAAJ>
  - [63] Peter J. Denning. 1968. Thrashing: Its Causes and Prevention. 33 (01 1968), 915–922.
  - [64] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 439–450.
  - [65] Kevin Comerford Kahn. 1976. *Program Behavior and Load Dependent System Performance*. Ph.D. Dissertation. West Lafayette, IN, USA. AAI7707474.
  - [66] S. Krumke. 2002. Online Optimization: Competitive Analysis and Beyond. 8.
  - [67] Redis Labs. 2014. Memtier BenchMark. [https://github.com/RedisLabs/memtier\\_benchmark](https://github.com/RedisLabs/memtier_benchmark) (Visited on 2021-May-24).

- [68] Jacques Lenfant. 1977. Comparison of the Working Sets and Bounded Locality Intervals of a Program. In *Proceedings of the International Workshop Organized by the Commission of the European Communities on Modelling and Performance Evaluation of Computer Systems*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 217–239. <http://dl.acm.org/citation.cfm?id=647407.724414>
- [69] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data> (Visited on 2021-May-24).
- [70] Jan Lindström, Vilho Raatikka, Jarmo Ruuth, Petri Soini, and Katriina Vakkila. 2013. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. *IEEE Data Eng. Bull.* 36 (01 2013), 14–20.
- [71] Linux. None. OOM Killer Working. [https://linux-mm.org/OOM\\_Killer](https://linux-mm.org/OOM_Killer). (Visited on 2021-May-24).
- [72] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. 2012. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon) (ISCA '12). IEEE Computer Society, Washington, DC, USA, 1–12. <http://dl.acm.org/citation.cfm?id=2337159.2337161>
- [73] ARM Ltd. 2018. *ARM CoreLink™ MMU-500 System Memory Management Unit*.
- [74] A. Wayne Madison and Alan P. Batson. 1976. Characteristics of Program Localities. *Commun. ACM* 19, 5 (May 1976), 285–294. <https://doi.org/10.1145/360051.360227>
- [75] Yuki Matsubara and Yukinori Sato. 2014. Online Memory Access Pattern Analysis on an Application Profiling Tool. In *2014 Second International Symposium on Computing and Networking*. 602–604. <https://doi.org/10.1109/CANDAR.2014.86>
- [76] R. L. Mattson, J. Gececi, D. R. Slutz, and I. L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Syst. J.* 9, 2 (June 1970), 78–117. <https://doi.org/10.1147/sj.92.0078>
- [77] Wolfgang Maier. 2008. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK.
- [78] Jim Mauro and Richard McDougall. 2001. *Solaris Internals: Core Kernel Architecture*. Sun Microsystems, Inc., Mountain View, CA, USA.
- [79] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. 2014. *The Design and Implementation of the FreeBSD Operating System* (2nd ed.). Addison-Wesley Professional.
- [80] D. C. Montgomery and G. C. Runger. 2003. *Applied Statistics and Probability for Engineers*. John Wiley and Sons.
- [81] Satoru Moriya. 2011. Individual Tunable Watermark. (2011). <https://lwn.net/Articles/422291/> (Visited on 2021-May-24).
- [82] Yoshihiro Oyama, Shun Ishiguro, Jun Murakami, Shin Sasaki, Ryo Matsumiya, and Osamu Tatebe. 2014. Reduction of Operating System Jitter Caused by Page Reclaim. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers* (Munich, Germany) (ROSS '14). ACM, New York, NY, USA, Article 9, 8 pages. <https://doi.org/10.1145/2612262.2612270>
- [83] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisirioj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. [n.d.]. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores.
- [84] Kishore Kumar Pusukuri. 2014. Working Set Model for Multithreaded Programs. In *Proceedings of the 2014 International Conference on Timely Results in Operating Systems* (Broomfield, CO) (TRIOS '14). USENIX Association, Berkeley, CA, USA, 1–1. <http://dl.acm.org/citation.cfm?id=2750315.2750316>
- [85] Vineet Sajwan, Varnita Yadav, and M. Haider. 2015. The Hadoop Distributed File System: Architecture and Internals. *International Journal of Combined Research & Development (IJCRD)* pISSN (05 2015), 2321–2241.
- [86] ServerFault. 2012. How do I prevent Linux from freezing when out of memory? <https://serverfault.com/questions/390623/how-do-i-prevent-linux-from-freezing-when-out-of-memory>. (Visited on 2021-May-24).
- [87] Samuel Sanford Shapiro and Martin B Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* 52, 3/4 (1965), 591–611.
- [88] Xipeng Shen, Yutao Zhong, and Chen Ding. 2004. Locality Phase Prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, USA) (ASPLOS XI). Association for Computing Machinery, New York, NY, USA, 165–176. <https://doi.org/10.1145/1024393.1024414>
- [89] T. Sherwood, E. Perelman, and B. Calder. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. 3–14. <https://doi.org/10.1109/PACT.2001.953283>
- [90] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. *SIGPLAN Not.* 37, 10 (Oct. 2002), 45–57. <https://doi.org/10.1145/605432.605403>
- [91] Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase Tracking and Prediction. *SIGARCH Comput. Archit. News* 31, 2 (May 2003), 336–349. <https://doi.org/10.1145/871656.859657>
- [92] J.E. Smith and A.S. Dhodapkar. 2002. Dynamic microarchitecture adaptation via co-designed virtual machines. In *2002 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No.02CH37315)*. Vol. 1. 198–199 vol.1. <https://doi.org/10.1109/ISSCC.2002.993004>
- [93] J. Spirn. 1976. Distance String Models for Program Behavior. *Computer* 9, 11 (Nov 1976), 14–20. <https://doi.org/10.1109/C-M.1976.218435>
- [94] Clayton Thomas and Bob Sheldon. 1999. The “Knee of a Curve”— Useful Clue but Incomplete Support. *Military Operations Research* 4, 2 (1999), 17–24. <http://www.jstor.org/stable/43940795>
- [95] Rollins Turner and Bill Strecker. 1977. Use of the LRU Stack Depth Distribution for Simulation of Paging Behavior. *Commun. ACM* 20, 11 (Nov. 1977), 795–798. <https://doi.org/10.1145/359863.359868>
- [96] Jakob Unterwurzacher. 2017. earlyoom - The Early OOM Daemon. <https://github.com/rfjakob/earlyoom>. (Visited on 2021-May-24).
- [97] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) (FAST'15). USENIX Association, Berkeley, CA, USA, 95–110. <http://dl.acm.org/citation.cfm?id=2750482.2750490>
- [98] Johannes Weiner. 2018. Tracking pressure-stall information. (2018). <https://lwn.net/Articles/759781/> (Visited on 2021-May-24).
- [99] Gerhard Wiesinger. 2016. Still OOM problems with 4.9er kernels. <https://lkml.org/lkml/2016/12/10/73/>. (Visited on 2021-May-24).
- [100] David P. Williamson and David B. Shmoys. 2011. *The Design of Approximation Algorithms*. Cambridge University Press. <https://doi.org/10.1017/CBO9780511921735>
- [101] Paul R. Wilson. 2007. Locality of Reference, Patterns in Program Behavior, Memory Management, and Memory Hierarchies.
- [102] X. Xiang, B. Bao, C. Ding, and Y. Gao. 2011. Linear-time Modeling of Program Working Set in Shared Cache. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 350–360. <https://doi.org/10.1109/PACT.2011.66>
- [103] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: A Higher Order Theory of Locality. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) (ASPLOS '13). ACM, New York, NY, USA, 343–356. <https://doi.org/10.1145/2451116.2451153>
- [104] Daniel Xu. 2018. Open-sourcing oomd, a new approach to handling OOMs. <https://engineering.fb.com/production-engineering/oomd/>. (Visited on 2021-May-24).
- [105] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) (HotCloud '10). USENIX Association, USA, 10.
- [106] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1920–1948. <https://doi.org/10.1109/TKDE.2015.2427795>
- [107] Weiming Zhao, Xinxin Jin, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Xiaoming Li. 2011. Low Cost Working Set Size Tracking. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Portland, OR) (USENIXATC'11). USENIX Association, Berkeley, CA, USA, 17–17. <http://dl.acm.org/citation.cfm?id=2002181.2002198>
- [108] Weiming Zhao and Zhenlin Wang. 2009. Dynamic Memory Balancing for Virtual Machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Washington, DC, USA) (VEE '09). ACM, New York, NY, USA, 21–30. <https://doi.org/10.1145/1508293.1508297>
- [109] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program Locality Analysis Using Reuse Distance. *ACM Trans. Program. Lang. Syst.* 31, 6, Article 20 (Aug. 2009), 39 pages. <https://doi.org/10.1145/1552309.1552310>
- [110] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program Locality Analysis Using Reuse Distance. *ACM Trans. Program. Lang. Syst.* 31, 6, Article 20 (Aug. 2009), 39 pages. <https://doi.org/10.1145/1552309.1552310>
- [111] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuan Yuan Zhou, and Sanjeev Kumar. 2004. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, USA) (ASPLOS XI). ACM, New York, NY, USA, 177–188. <https://doi.org/10.1145/1024393.1024415>