AEQUAMEM: Why Reclaim at Local Limits When You Can Borrow Idle Capacity from Other Tenants?

Yunfei Zhang[‡,§], Wenda Tang[§,*], Jianjiang Li[‡,*], Jie Wu[§,¶]
[‡]University of Science and Technology Beijing, Beijing, China
[§]China Telecom Cloud Computing Research Institute, Beijing, China
[¶]Temple University, Philadelphia, PA, USA
Email: zhangyunfei@xs.ustb.edu.cn, tangwd1@chinatelecom.cn, lijianjiang@ustb.edu.cn, jiewu@temple.edu

***Abstract*—Memory cgroups isolate tenants with fixed memory limits, but this isolation often lowers utilization when time-varying demand leaves capacity stranded in idle tenant cgroups while active tenants experience memory pressure. Although temporarily reallocating idle capacity to under-pressure cgroups can improve utilization, it weakens explicit cgroup boundaries and can make borrowed memory costly to reclaim when demand shifts. We present AEQUAMEM, a user-space memory controller that lends unused headroom in idle tenant cgroups to tenants under memory pressure. AEQUAMEM uses Linux Pressure Stall Information (PSI) to identify pressured tenants and increases their memory limits only by headroom harvested from idle cgroups, keeping the total memory budget unchanged. It maintains a credit ledger to account for donation history across workload phases. When demand exceeds available headroom, AEQUAMEM biases allocation toward tenants with more file-backed footprints, since reducing their leases is less disruptive when demand shifts. AEQUAMEM requires no kernel changes, making it deployable in existing cloud environments. In synthetic experiments with imbalanced memory demand, AEQUAMEM avoids the task failures observed under the Static fixed-limit policy and reduces file-backed workload completion time by up to $9\times$; in bursty trace-driven experiments, it maintains a Jain's fairness index above 0.91 and achieves up to $2.2\times$ aggregate speedup over the Static policy.**

***Index Terms*—Elastic memory management, Linux control groups (cgroups), multi-tenancy, pressure stall information (PSI).**

I. INTRODUCTION

Cloud platforms increasingly multiplex physical resources across containers to improve utilization and reduce operational cost [1]–[3]. Linux control groups (cgroups) provide the kernel interface for isolating those containers [4]. For memory, operators commonly set each tenant a fixed limit such as `memory.max`. Fixed limits are predictable, but they cannot adapt to memory demand that varies across tenants and over time. This can cause unavoidable reclaim stalls and leaves memory stranded in underutilized cgroups [5]–[8]. Figure 1 illustrates this quota imbalance: one tenant reaches its local limit while another leaves quota headroom unused, forcing further demand to trigger local reclaim.

However, relaxing these limits without careful coordination would compromise performance isolation, as one tenant could consume memory needed by others. An ideal controller should therefore redistribute idle memory across tenants while respecting the total budget and promptly reclaiming borrowed memory when the reserving tenant needs it. Furthermore, because additional memory can improve application performance, limited idle capacity should be allocated fairly among competing tenants.

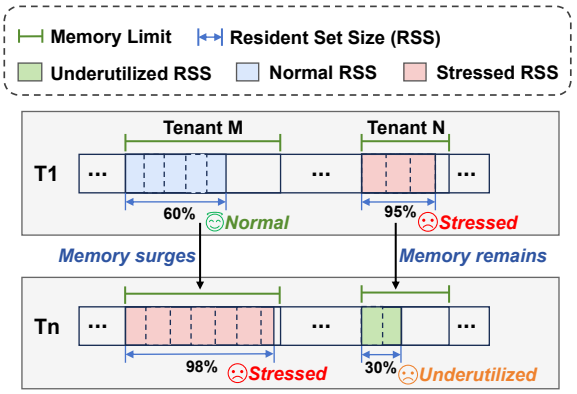


Fig. 1. Static limits strand memory when one tenant reaches its local limit while another leaves quota headroom unused.

Meeting these requirements simultaneously poses three challenges. The controller must ❶ determine when to redistribute idle memory, ❷ allocate limited capacity fairly, and ❸ minimize the reclaim cost caused by later quota reductions.

Existing systems address complementary parts of these challenges. Karma uses credits to allocate resources under time-varying demand [9], while Autopilot adjusts task concurrency and resource limits from historical workload observations [10]. Other memory managers reduce interference or improve capacity use through cache partitioning, NUMA placement, and tiered-memory migration [11]–[17]. These mechanisms primarily decide where pages reside or move, rather than how a fixed aggregate memory budget should be lent among tenant cgroups. Consequently, existing approaches do not jointly provide pressure-based demand detection, history-aware quota allocation, and reclaim-cost-aware borrower ordering.

To address this gap, we present AEQUAMEM[1], a user-space controller for bounded memory lending within a fixed aggregate memory budget. AEQUAMEM uses Linux's Pressure Stall Information (PSI) to identify borrowers experiencing measurable memory-pressure stalls and compares each

[*]Corresponding authors.

[1]The name AEQUAMEM derives from the Latin *aequitas* (equity or fairness), reflecting its goal of fair and flexible memory sharing across tenants.

979-8-3195-9980-3/26/$31.00 ©2026 IEEE

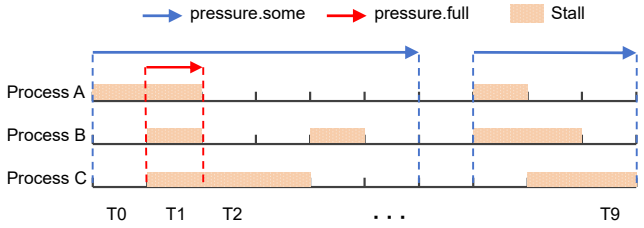


Fig. 2. Conceptual calculation of PSI `some` and `full`.

cgroup's current usage with its limit to identify unused donor capacity. A credit ledger accounts for prior donations when tenants compete for limited capacity, while memory composition prioritizes borrowers with lower expected reclaim cost. Operating as a closed loop over existing cgroup v2 interfaces, AEQUAMEM requires neither workload instrumentation nor kernel modification.

This paper makes the following contributions:

- **Empirical Characterization.** We characterize the isolation–elasticity trade-off of cgroup memory limits and show that controlled quota redistribution can relieve local memory pressure without expanding the aggregate memory budget. We show that file-backed and anonymous-heavy workloads incur substantially different reclaim costs, motivating composition-aware borrower ordering.
- **AEQUAMEM Design.** We design AEQUAMEM, a user-space controller for bounded memory lending within a fixed aggregate memory budget. It uses PSI to identify borrowers experiencing measurable memory-pressure stalls and maintains a credit ledger to track donation history across workload phases. When multiple borrowers compete for limited capacity, it uses memory composition to prioritize those with lower expected reclaim cost. AEQUAMEM operates through existing cgroup v2 interfaces without workload instrumentation or kernel modification.
- **Evaluation.** We evaluate AEQUAMEM across four controlled resource states and bursty trace-driven scenarios. In the controlled experiments, AEQUAMEM avoids the task failures observed under Static and achieves a speedup of up to 9.2× for file-backed workloads. Under moderate and high contention, it achieves 2.2× and 1.9× aggregate speedup over Static, respectively, while maintaining JFI above 0.91.

II. Background and Motivation

A. Memory Isolation and Pressure

Cgroup v2 exposes `memory.max` as a hard limit and `memory.high` as a throttling boundary [4]. Near these limits, the kernel reclaims pages inside the cgroup. File-backed pages may be evicted and re-read, whereas anonymous pages require swap space if they are to survive reclamation. Failed reclaim can end in an out-of-memory (OOM) kill.

Reclaim also consumes processor time and can block the allocating task. The kernel scans the cgroup's least-recently-used lists, evicts clean cache pages, writes dirty pages when required, and may swap anonymous pages. If this work runs in direct reclaim, application progress waits for enough memory

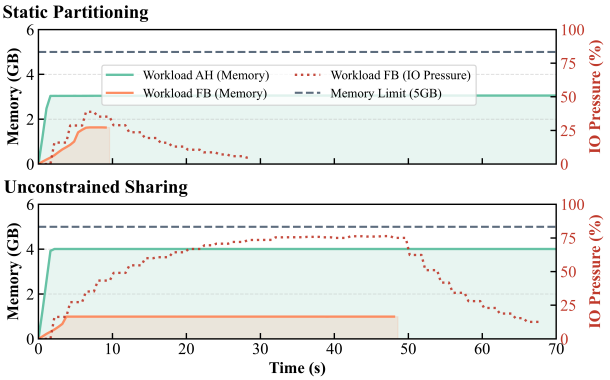


Fig. 3. Per-tenant limits protect FB from an aggressive AH neighbor; removing them causes cache thrashing and a 5.7× slowdown.

to become available. A memory limit can therefore harm performance before an OOM kill occurs. Raw usage shows proximity to the limit, but it does not show how much execution time the reclaim path has already consumed.

Pressure Stall Information (PSI) reports time lost because tasks wait for CPU, memory, or I/O [18]. For memory, `some` records intervals in which at least one task stalls, while `full` records intervals in which all non-idle tasks stall. In Fig. 2, at least one task stalls in nine of ten intervals, and all tasks stall in one, producing `some`=90% and `full`=10%. We use PSI as a demand signal because it measures stall exposure rather than allocation alone.

B. Isolation, Stranding, and Reclaim Cost

We use a dual-tenant cgroup hierarchy on an Intel Xeon Platinum 8566C host with 128 GB DRAM and Linux 6.8. An experiment-wide cgroup enforces an aggregate memory budget of 4–5 GB. The workloads are either anonymous-heavy (AH), allocating and touching anonymous pages, or file-backed (FB), scanning a fixed corpus whose active footprint is mainly page cache. We report execution time, memory usage, and PSI.

Isolation prevents monopolization. We place a 4 GB AH workload and a 1.7 GB FB working set within a 5 GB aggregate memory budget. With tenant-cgroup limits of 3 GB for AH and 2 GB for FB, the FB task completes in 8.5 s. Without per-tenant quota isolation, AH grows to 4 GB and leaves about 1 GB for FB. Cache thrashing then raises FB time to 48.1 s and keeps I/O pressure above 50% for more than half of the run (Fig. 3). Elasticity must therefore retain explicit per-tenant boundaries.

Static limits strand usable capacity. We next assign 4 GB to an AH tenant that uses 1 GB and 1 GB to an FB tenant with a 1.7 GB working set. At the static limit, FB takes more than 45 s. Reassigning 20% of AH's idle reservation raises FB's available memory to about 1.6 GB and reduces its time to 13 s (Fig. 4). Thus, controlled limit changes can relieve local pressure without exceeding the fixed aggregate budget.

Reclaim cost after quota reduction depends on memory composition. In a 4 GB pool, a borrower grows to about 3.3 GB before the lender requests capacity at $t = 6$ s.

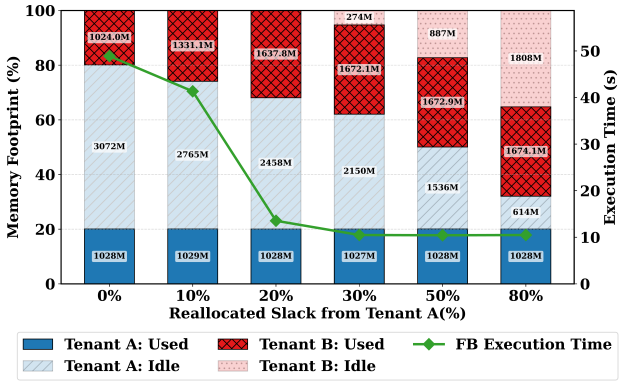


Fig. 4. Reallocating part of AH's idle reservation relieves FB's local scarcity.

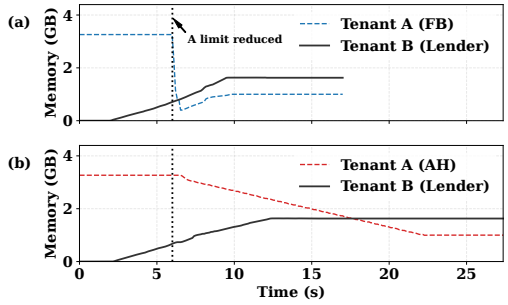


Fig. 5. The lender completes later when the same quota reduction is applied to an AH rather than an FB borrower.

Reducing an FB borrower's quota lets the lender finish in 10.0 s; applying the same reduction to an AH borrower increases its time to 12.5 s (Fig. 5).

A separate 3 GB-to-1 GB quota reduction explains the gap (Fig. 6). The FB footprint drops to 0.4 GB within 0.5 s, while AH takes about 14 s to reach 1 GB. AH memory pressure peaks near 70% and persists for almost 20 s; FB remains below 3%. The contrast establishes memory composition as a useful predictor of reclaim cost for these workloads and motivates the ordering policy.

The kernel paths explain why the two traces diverge. Clean file-backed pages already have persistent copies, so the kernel can discard page-cache entries and reduce the cgroup footprint without first writing their contents elsewhere. A later access may incur a refault, but the resulting reclaim can complete quickly. Anonymous pages have no file backing. Preserving them requires allocating swap space, updating page tables, and writing page contents before memory can be released. When the reclaim runs synchronously, the application waits for this work and PSI rises. If swap is unavailable or reclaim cannot free enough memory, the kernel may instead invoke the OOM path.

The traces therefore show slower AH memory reduction under our no-swap condition than eviction of mostly clean FB cache. Composition is a reclaim-cost heuristic, not a guarantee: dirty file pages may require writeback, while swap or compression may lower anonymous reclaim cost. AEQUAMEM therefore re-evaluates composition and pressure after every control cycle.

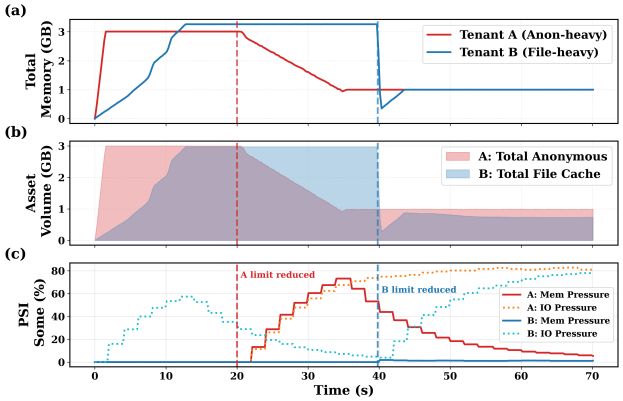


Fig. 6. Reclaim behavior after the same 3 GB-to-1 GB limit reduction: footprint, memory composition, and PSI response for FB and AH borrowers.

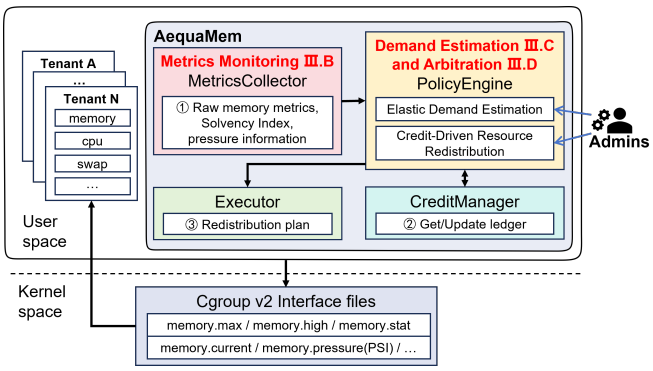


Fig. 7. AEQUAMEM control path for bounded cgroup quota actuation.

III. DESIGN OF AEQUAMEM

A. Architecture and Control Path

AEQUAMEM turns fixed tenant-cgroup limits into bounded quota leases within an administrator-configured aggregate memory budget. Its control policy uses PSI to admit borrowers with measurable memory-pressure stalls, the usage-to-limit ratio to identify donor headroom, and donation history to determine credit-based priority. When eligible borrowers compete for limited capacity, memory composition provides a heuristic for ordering them by expected reclaim cost. The aggregate budget remains fixed: AEQUAMEM neither creates capacity beyond it nor permanently changes tenant allocations. Instead, each cycle converts verified idle headroom into a logical pool, constructs a complete plan from a common snapshot, and re-evaluates active leases as workload phases change.

Figure 7 shows the closed loop. The `MetricsCollector` reads PSI, memory usage, limits, and memory composition. The `PolicyEngine` identifies donors and pressured borrowers, estimates expansion, and constructs a reallocation plan. The `CreditManager` records donations and grants. Finally, the `Executor` applies the plan through cgroup v2 `memory.max`. A whitelist confines control to a configured set of cgroups.

Algorithm 1: Heuristic Demand Estimation

Input : Metrics M_i, solvency $\mathcal{S}_i$
Output: Target limit $D_{i,\text{target}}$

$W_{i,\text{dom}} \leftarrow M_{i,\text{file}}$ if $\mathcal{S}_i > 0.5$, else $M_{i,\text{anon}}$;
$\gamma_i \leftarrow \text{MapPSIToScale}(M_{i,\text{psi}})$;
$D_{i,\text{target}} \leftarrow \gamma_i W_{i,\text{dom}}$;
return $D_{i,\text{target}}$;

B. Monitoring Pressure and Memory Composition

The collector samples PSI, memory usage, limits, and composition at each interval. It obtains them from the cgroup v2 pressure, current-usage, limit, and memory-statistics interfaces. PSI indicates whether the current limit is stalling progress. For tenant i, we summarize composition using

$$\mathcal{S}_i = \frac{M_{i,\text{file}}}{M_{i,\text{anon}} + M_{i,\text{file}}}. \tag{1}$$

A larger $\mathcal{S}_i$ indicates a more file-backed footprint in the current sample. The signals remain separate: PSI admits borrowers, the usage-to-limit ratio identifies donor slack, and composition orders eligible borrowers. Motivated by Sec. II-B, the policy uses composition as a reclaim-cost heuristic; the next sample updates the estimate as writeback, refaults, or swap activity changes the footprint.

C. Pressure-Gated Demand Estimation

When $\text{PSI}_{\text{some}} > \epsilon_{\text{psi}}$, the controller estimates an expansion from the dominant memory component. It selects file usage when $\mathcal{S}_i > 0.5$ and anonymous usage otherwise:

$$W_{i,\text{dom}} = \begin{cases} M_{i,\text{file}}, & \mathcal{S}_i > 0.5, \\ M_{i,\text{anon}}, & \text{otherwise.} \end{cases} \tag{2}$$

The target is $D_{i,\text{target}} = \gamma_i W_{i,\text{dom}}$. The evaluated mapping uses $\gamma_i = 1.2$, 1.3, and 1.5 for PSI at most $3\epsilon_{\text{psi}}$, between $3\epsilon_{\text{psi}}$ and $10\epsilon_{\text{psi}}$, and above $10\epsilon_{\text{psi}}$, respectively. Algorithm 1 summarizes the calculation, and each grant is capped by capacity harvested in the same cycle.

D. Credit- and Composition-Aware Arbitration

The credit ledger adapts Karma's gift-exchange principle to cgroup memory quotas [9]. A *borrower* exceeds ϵ_{psi}, whereas a *donor* remains below ϵ_{util}. Every one-second cycle updates tenant i by $-(L_i - L_i^0)/100$ credits, with limits measured in MiB; a tenant below its initial limit earns credit, while one above it spends credit. Donated capacity enters a logical pool. Borrowers are ordered by credit and then PSI before the solvency partition; tenant identifiers provide deterministic residual ordering. Unlike published Karma, this controller adds pressure sensing, composition-aware ordering, and direct cgroup actuation.

Algorithm 2 shows one control cycle. It separates donor selection from borrower allocation, caps every grant by the available pool, and stops when the pool is exhausted. The executor then applies the planned limits. Credit supplies history-aware ordering, while under contention, the solvency partition places borrowers that can return memory more quickly ahead of the others.

Algorithm 2: Credit-Based Memory Arbitration

Input : Tenants $\mathcal{T}$, credits C, metrics M, thresholds $\epsilon_{\text{psi}}, \epsilon_{\text{util}}$
Output: Limit-update plan $\mathcal{P}$

$Pool \leftarrow \sum_{i \in \mathcal{T}} \text{Scavenge}(i)$ for $M_i.usage/M_i.limit < \epsilon_{\text{util}}$;
$\mathcal{B} \leftarrow \{i \in \mathcal{T} : M_{i,\text{psi}} > \epsilon_{\text{psi}}\}$;
Sort $\mathcal{B}$ by descending credit and pressure;
Stable-partition $\mathcal{B}$ by $\mathcal{S} > 0.5$ first;
foreach $b \in \mathcal{B}$ **do**
  $D \leftarrow \text{EstimateDemand}(M_b, \mathcal{S}_b) - M_b.limit$;
  $G \leftarrow \min(\max(D, 0), Pool)$;
  $\mathcal{P} \leftarrow \mathcal{P} \cup \text{UpdateLimit}(b, M_b.limit + G)$;
  $Pool \leftarrow Pool - G$; **if** $Pool = 0$ **then**
    break
return $\mathcal{P}$;

The separation of signals also defines the ablation boundary. PSI determines whether a tenant enters the borrower set; it does not rank an unpressured tenant merely because its usage is high. Credit orders borrowers according to prior contributions, while solvency partitions that ordered list by expected reclaim cost inferred from current composition. AEQUAMEM-NoSolv removes only this final partition while retaining the same demand detector, credit ledger, donor pool, and executor. Differences from the full policy therefore isolate composition-aware ordering.

E. Control Cycle

Each control cycle separates observation, planning, and actuation. The collector first obtains one consistent set of cgroup counters for every managed tenant. The policy then labels tenants without changing any limit. Only after it has identified the donor pool and ordered all borrowers does the executor issue updates. This separation prevents an early update from changing the measurements used to rank later tenants in the same cycle.

Because every borrower is compared from the same snapshot, policy priority is independent of executor write order and reflects the state observed at the sampling boundary.

Donor capacity is computed from the gap between current use and limit. The evaluated policy uses $\epsilon_{\text{util}} = 0.7$, harvests half of eligible headroom, and never shrinks a donor below 20% of its initial limit. It aggregates donations rather than binding one donor to one borrower, without changing the aggregate memory budget.

The credit ledger records the duration and magnitude of donations. A tenant therefore gains priority by leaving capacity available over time, not merely by appearing idle in one sample. When several borrowers cross the PSI threshold, the policy orders them by credit and pressure. It then stable-partitions that list into high-solvency and low-solvency groups.

Solvency therefore determines the primary group, while the prior order is preserved within each group. A grant decreases the remaining pool, and the cycle terminates when either all requests are served or no capacity remains.

The executor applies every planned limit update through the tenant cgroup's `memory.max` interface. The next sample observes the kernel's response, including any residual PSI and changes in anonymous/file composition. The design therefore forms a feedback controller without requiring an application API or kernel patch. It does not bypass kernel reclamation: an aggressive reduction can still stall or kill a tenant if the selected limit is infeasible. In practice, the quality of a plan depends on the PSI threshold, utilization threshold, expansion mapping, and sampling period evaluated in Sec. IV.

F. Assumptions and Scope

AEQUAMEM assumes that the managed tenant cgroups share an administrative trust domain and operate within a fixed aggregate memory budget that defines the redistribution boundary. It also assumes that current memory composition is informative about near-term reclaim behavior after a quota reduction. This assumption is supported by Fig. 5 and Fig. 6, but it is not exact. Since dirty file pages requires write-back, swap or compression lowers anonymous reclaim cost, and composition changes between samples, composition only guides borrower ordering instead of guaranteeing reclaim cost.

Credit captures donation history, whereas solvency summarizes the current anonymous/file composition. AEQUAMEM stably partitions borrowers by composition while preserving credit order within each group. Its binary threshold may rank a marginally file-dominant borrower above a highly credited anonymous-dominant one; alternative scores may yield different fairness–reclaim-cost trade-offs.

G. End-to-End Control Walkthrough

Consider one cycle with several managed tenants. The collector first reads each tenant's PSI, current usage, limit, anonymous/file breakdown, and credit balance. A tenant whose PSI remains below ϵ_{psi} is not treated as a borrower, even if it is close to its limit. This rule avoids granting memory solely because a workload occupies its reservation. Conversely, a tenant becomes a borrower when its limit produces measurable stalls, even when its raw usage changes little.

For every underutilized tenant, the policy computes capacity that can be removed without crossing the configured utilization boundary. These contributions form $Pool$. The corresponding donation is recorded in the credit ledger before borrower selection. If $Pool = 0$, the cycle produces no expansion plan; PSI alone does not authorize the controller to exceed the fixed aggregate memory budget. This case separates demand detection from feasible allocation.

The policy next evaluates the borrowers. Algorithm 1 converts the dominant memory component and PSI band into a target limit. Subtracting the current limit gives the unsatisfied request. Borrowers are sorted by credit and pressure, then stable-partitioned by $\mathcal{S} > 0.5$. The first borrower receives $\min(D, Pool)$. Any residual pool is offered to the next borrower until either the requests or the pool are exhausted.

The executor applies only the completed plan. Because each grant is bounded by harvested capacity, the total planned quota expansion across tenant cgroups cannot exceed the pool created in that cycle. The enclosing cgroup enforces the fixed aggregate memory budget as a second bound. The next monitoring sample then closes the loop: falling PSI indicates that pressure has decreased under the new limit, while persistent PSI keeps the tenant eligible for another bounded adjustment. A change in memory composition also changes the tenant's solvency rank in the next cycle.

Three boundary cases remain. Global scarcity can leave the pool empty or insufficient; composition can change between samples; and a donor can resume activity after its limit is reduced. Bounded grants and repeated observation limit these risks while preserving the aggregate memory budget, although the current prototype provides neither transactional rollback nor a worst-case bound on lease-withdrawal latency.

H. Complexity Analysis and Default Parameters

Let n be the number of managed cgroups and $b \leq n$ the number of borrowers in one cycle. Metric collection and donor identification each scan the tenant set once, requiring $O(n)$ controller operations in addition to kernel-interface reads. Demand estimation is constant work per borrower. Sorting by credit and pressure costs $O(b \log b)$, and the solvency partition and grant construction each scan the ordered borrowers once. The controller-side complexity of one cycle is therefore $O(n + b \log b)$ with $O(n)$ state for metrics and credit balances. This analysis excludes the kernel cost of reclaiming pages after a limit change, which can dominate the controller computation and is precisely the cost that composition-aware ordering seeks to reduce.

The four main parameters serve distinct purposes. The sampling period controls observation latency, ϵ_{psi} separates occupancy from visible demand, ϵ_{util} controls harvesting, and the PSI-to-γ mapping determines request size. We use a one-second period, $\epsilon_{\text{psi}} = 0.01$, and $\epsilon_{\text{util}} = 0.7$; deployments should tune these controls independently.

IV. EVALUATION

Our evaluation answers two questions: (RQ1) Can dynamic limits recover locally stranded memory? (RQ2) Does solvency-aware ordering improve pressure- and credit-driven arbitration under bursty contention?

A. Methodology

Platform. Experiments run on an Intel Xeon Platinum 8566C server with 16 physical cores and 128 GB DDR5, Ubuntu 24.04, and Linux 6.8.0. For controlled comparison, we disable SMT and swap, pin each tenant to a CPU, and drop page caches before each run. These controls reduce interference between tenants and give every FB run the same cold-cache starting condition. AEQUAMEM is a Python user-space daemon that reads cgroup v2 and PSI interfaces directly.

TABLE I
CONTROLLED TWO-TENANT SCENARIOS.

ID	Workload mix	Resource state
S_1	Idle AH + Idle FB	Total $>$ peak
S_2	AH + Idle FB	A below demand
S_3	Idle AH + FB	B below demand
S_4	AH + FB	Total $<$ peak

Workloads. The AH workload uses GAPBS [19]; its graph kernels generate an anonymous-memory-dominant footprint with irregular accesses. The FB workload scans a large file corpus and populates the page cache. These endpoints isolate distinct reclaim paths while exposing comparable completion-time and PSI measurements; mixed production footprints are outside this controlled evaluation.

Policies. *Static* retains each tenant cgroup's initial limit. AEQUAMEM-NoSolv is the Karma-inspired no-solvency ablation of AEQUAMEM [9]: it retains PSI admission, demand estimation, credit accounting, and cgroup actuation while removing the solvency partition. *Peak-memo* applies exponential decay and a 15% safety margin to recent peaks, following Autopilot's moving-window intuition [10]. AEQUAMEM is the full policy in Sec. III. Every dynamic policy runs once per second. We use $\epsilon_{\text{psi}} = 0.01$ and initialize each tenant with 10^6 credits.

A common controller framework holds monitoring, actuation, and sampling constant across the dynamic policies. Static provides the hard-limit reference point, Peak-memo provides a history-based local heuristic, and the AEQUAMEM-NoSolv–AEQUAMEM comparison isolates the solvency partition while holding the rest of the control path fixed.

Comparison discipline. The dynamic policies consume the same cgroup counters, run at the same period, and write limits through the same executor. Their differences therefore isolate allocation rules: Peak-memo tests history-based estimation, while the AEQUAMEM-NoSolv–AEQUAMEM comparison isolates composition-aware ordering.

B. RQ1: Recovering Stranded Memory

Table I defines four two-tenant states. S_1 has enough capacity for both peaks; S_2 and S_3 constrain one tenant while the other is idle; and S_4 constrains their combined demand.

We divide the pairwise experiments into four controlled cases. S_1 verifies that a controller does not perturb workloads whose initial quotas already satisfy demand. S_2 and S_3 reverse the pressured tenant, testing idle-reservation transfers in both AH-to-FB and FB-to-AH directions. S_4 creates genuine scarcity by making the combined peak demand infeasible. Together, these cases cover sufficient memory, recoverable local stranding, and total demand above the fixed budget.

Table II uses red for a failed task or severe Static degradation, orange for completion with one OOM event, and blue for dynamic-policy completion that avoids the corresponding Static failure or slowdown.

All policies behave similarly in S_1. Static fails the AH task in S_2 and S_4, and stretches FB completion to about 202 s in S_3 and S_4. Each dynamic policy transfers idle capacity and completes both workloads, although all three record one OOM event for AH in S_4. For FB, AEQUAMEM completes in 24.07 s and 22.07 s in S_3 and S_4, versus 201.77 s and 202.18 s under Static. AEQUAMEM-NoSolv records similar times, while Peak-memo reaches 24.80 s and 24.70 s. RQ1 therefore confirms that dynamic redistribution recovers locally stranded capacity in these states. The pairwise cases do not separate AEQUAMEM from AEQUAMEM-NoSolv because they seldom present competing borrowers for solvency ordering.

The four states also expose the boundary of redistribution. In S_2 and S_3, one idle tenant supplies enough memory for its active neighbor. In S_4, combined peak demand exceeds the fixed aggregate memory budget, so every controller must arbitrate rather than eliminate scarcity. The OOM event observed by each dynamic policy confirms that quota adjustment allows more tasks to continue or complete but cannot create capacity.

The FB results quantify the cost of leaving this imbalance unresolved. In S_3, AEQUAMEM reduces completion time from 201.77 s to 24.07 s, an $8.4\times$ speedup. In S_4, it reduces 202.18 s to 22.07 s, or $9.2\times$ speedup. The near-symmetric behavior of the dynamic policies in these pairwise states is informative: once an idle reservation is available and only one effective borrower needs it, lending dominates the result and the solvency partition has little opportunity to change the order.

C. RQ2: Borrower Ordering under Bursty Contention

We replay the same event timeline for every policy within each pressure regime. Each regime uses one fixed one-hour Poisson-arrival trace (seed 2027, 3 s mean interarrival time) with 1,142 events over 12 tenants: four AH and eight FB cgroups, each with one CPU and an initial 3 GB limit within a fixed 36 GB aggregate memory budget. The high-pressure trace expands the AH graph-intensity range from 18–21 to 18–23. Bars aggregate task outcomes within each fixed trace.

For task i, the Performance Maintenance Ratio (PMR) compares standalone and contended completion time. Jain's Fairness Index (JFI) summarizes the distribution of PMR [20]:

$$\text{PMR}_i = \frac{T_{i,\text{standalone}}}{T_{i,\text{actual}}}, \qquad \text{JFI} = \frac{(\sum_i \text{PMR}_i)^2}{n \sum_i \text{PMR}_i^2}. \quad (3)$$

PMR equals one when a task matches its standalone completion time and falls below one as contention increases its runtime. JFI then measures how evenly those normalized outcomes are distributed across tasks. The two metrics answer different questions: average PMR captures absolute performance retention, whereas JFI captures dispersion. Reporting both prevents a policy from appearing effective merely because it slows every tenant by a similar amount.

Figure 8 reports PMR 0.65 and JFI 0.85 for Static. Static still obtains a JFI of 0.85 because several tenants experience similarly poor performance. Peak-memo reaches PMR 0.77 and JFI 0.88. AEQUAMEM and AEQUAMEM-NoSolv both

TABLE II
TASK COMPLETION TIME IN THE FOUR CONTROLLED SCENARIOS.

Scenario	Workload	Static	AEQUAMEM-NoSolv	Peak-memo	AEQUAMEM
S_1	AH	23.79±0.01 s	23.72±0.03 s	23.77±0.03 s	23.76±0.03 s
	FB	23.40±0.04 s	23.47±0.03 s	23.59±0.04 s	23.54±0.04 s
S_2	AH	Failed	23.74±0.01 s	23.75±0.05 s	23.74±0.03 s
	FB	21.95±0.04 s	23.53±0.10 s	23.55±0.03 s	23.54±0.05 s
S_3	AH	23.48±0.04 s	23.76±0.03 s	23.75±0.03 s	23.78±0.02 s
	FB	201.77±0.66 s	23.98±0.03 s	24.80±0.04 s	24.07±0.07 s
S_4	AH	Failed	22.72±0.01 s (OOM=1)	22.81±0.07 s (OOM=1)	22.73±0.03 s (OOM=1)
	FB	202.18±1.33 s	22.04±0.04 s	24.70±0.06 s	22.07±0.04 s

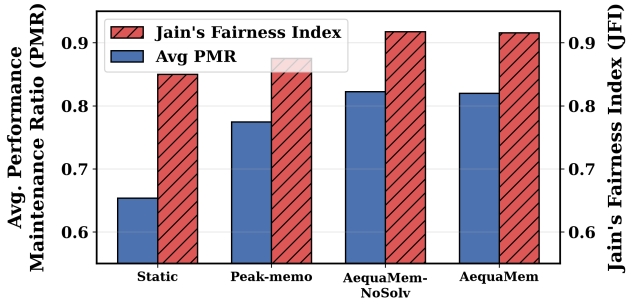


Fig. 8. Average PMR and JFI in the trace-driven experiment.

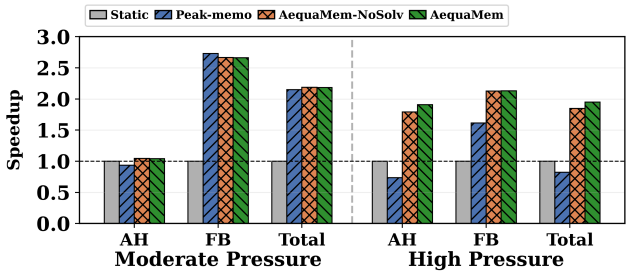


Fig. 9. Speedup over Static under moderate and high pressure.

exceed 0.91 JFI, consistent with their shared credit accounting and a more even distribution of normalized completion performance across tenants.

We compute speedup over Static from adjusted mean completion time. An OOM-failed task is assigned a multiplicative penalty greater than one relative to the longest successful duration for the same workload class and parameters, or relative to the class maximum when no exact match exists. Under moderate pressure, AEQUAMEM and AEQUAMEM-NoSolv both record 2.2× aggregate speedup, driven mainly by a 2.7× FB speedup (Fig. 9). Under high pressure, AEQUAMEM reaches 1.9×, compared with 1.8× for AEQUAMEM-NoSolv. Because solvency ordering is their only policy difference, the gap is consistent with its intended effect as contention intensifies. Peak-memo falls below 1× after the failure penalty.

The adjusted metric couples completion time with task survival instead of rewarding a policy that terminates difficult work. We therefore interpret speedup together with observed OOM outcomes. The high-pressure ablation isolates the incremental effect of solvency ordering under contention.

Overall, the credit-based policies achieve the highest JFI in the evaluated trace, while AEQUAMEM shows a small advantage over AEQUAMEM-NoSolv as contention intensifies.

V. DISCUSSION AND LIMITATIONS

Dynamic lending and solvency ordering. PSI-triggered lending provides most of the improvement over Static, while credit accounts for donation history across workload phases. Solvency addresses the narrower case in which multiple pressured borrowers compete for limited headroom. The high-pressure ablation indicates an incremental benefit from reclaim-cost-aware ordering under such contention.

Limits of the signals and metrics. The file-backed fraction is a reclaim-cost heuristic, not a universal cost model: dirty pages may require writeback, while swap or compression may lower anonymous reclaim cost. Our no-swap results represent a controlled AH setting rather than all production configurations.

JFI measures the distribution of per-task PMR, not tenant utility or SLO compliance, while adjusted completion time depends on the penalty assigned to failed tasks. We consequently interpret PMR, JFI, speedup, and OOM outcomes together. A broader sensitivity study should vary the PSI and utilization thresholds, sampling interval, and failure-penalty definition.

Evaluation and deployment scope. AEQUAMEM-NoSolv isolates the solvency partition, while Peak-memo represents history-based estimation. The evaluation covers AH and FB endpoints under fixed controller parameters. Future work will extend the evaluation to mixed footprints, NUMA-aware pools, and SLO-driven policies.

VI. RELATED WORK

Dynamic resource control. HarvestContainers reallocates spare CPU cores while protecting primary containers [21]; AgentCgroup targets tool-call-driven resource bursts in AI agents [22]. Autopilot adjusts task concurrency and resource limits from historical fleet observations [10]. Peak-memo adapts only Autopilot's historical-peak intuition as a local heuristic. AEQUAMEM instead arbitrates temporary quota loans among co-located cgroups, where temporary capacity must be returned within the fixed aggregate memory budget.

Credit-based allocation. Karma formalizes credit-based allocation for time-varying demands [9]. Thoth coordinates far-memory adjustments [23], while KarmaPM redistributes power budgets using donation history [24]. Memtrade harvests idle

memory by adjusting producer cgroup limits and matches it to remote consumers through a brokered marketplace [25]. These systems do not leverage anonymous/file-backed composition for reclaim-cost awareness in quota allocation; AEQUAMEM uses this distinction to order competing borrowers.

Memory capacity, placement, and reclamation. CXL supports coherent memory expansion and pooling [26], while TMO uses PSI to guide offloading to a slower tier [18]. NUMA and tiered-memory managers place or migrate pages using access frequency, recency, or migration cost [11]–[17]. These mechanisms determine where capacity comes from or where pages should reside. They can complement quota arbitration, but they do not remove per-tenant quota boundaries. AEQUAMEM operates at that boundary and allocates temporary shares of the fixed aggregate memory budget.

VII. CONCLUSION

AEQUAMEM enables tenant cgroups to share unused memory through bounded, reversible quota leases within a fixed aggregate memory budget. Its user-space control loop uses PSI to identify stalled borrowers, credits to track donation history, and memory composition to order borrowers only when demand exceeds available headroom. In controlled resource states, AEQUAMEM avoided the task failures observed under Static and achieved up to a $9\times$ speedup for file-backed workloads. Under trace-driven contention, it maintained JFI above 0.91 and achieved up to $2.2\times$ aggregate speedup over Static. These results show that pressure-gated, credit-backed quota arbitration recovers stranded memory without expanding that budget or requiring kernel changes.

ACKNOWLEDGMENT

We sincerely thank the anonymous reviewers for their constructive comments, which helped improve the presentation of this work. This work is supported by the National Natural Science Foundation of China (U25B2021).

REFERENCES

[1] D. O. Olabanji, O. O. Matthew, and T. Fitch, "Multi-tenancy in cloud-native architecture: a systematic mapping study," *WSEAS Transactions on Computers*, vol. 22, no. 4, pp. 25–43, 2023.

[2] V. Narasayya and S. Chaudhuri, "Multi-tenant cloud data services: State-of-the-art, challenges and opportunities," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 2465–2473.

[3] E. Truyen, D. Van Landuyt, V. Reniers, A. Rafique, B. Lagaisse, and W. Joosen, "Towards a container-based architecture for multi-tenant saas applications," in *Proceedings of the 15th international workshop on adaptive and reflective middleware*, 2016, pp. 1–6.

[4] The Linux Kernel development community, "Control group v2 — The Linux Kernel documentation," https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html, 2024, accessed: 2025-03-05.

[5] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica, "Jiffy: Elastic far-memory for stateful serverless analytics," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 697–713.

[6] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes, "Building an elastic query engine on disaggregated storage," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 449–462.

[7] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020, pp. 191–208.

[8] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter *et al.*, "The {CacheLib} caching engine: Design and experiences at scale," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 753–768.

[9] M. Vuppalapati, G. Fikioris, R. Agarwal, A. Cidon, A. Khandelwal, and É. Tardos, "Karma: Resource allocation for dynamic demands," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 645–662.

[10] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[11] J. Kim, P. Shin, S. Noh, D. Ham, and S. Hong, "Reducing memory interference latency of safety-critical applications via memory request throttling and linux cgroup," in *2018 31st IEEE International System-on-Chip Conference (SOCC)*. IEEE, 2018, pp. 215–220.

[12] S. Rai and M. Chaudhuri, "Using criticality of gpu accesses in memory management for cpu-gpu heterogeneous multi-core processors," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 1–23, 2017.

[13] H. Yun, W. Ali, S. Gondi, and S. Biswas, "Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms," *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1247–1252, 2016.

[14] H. Usui, L. Subramanian, K. K.-W. Chang, and O. Mutlu, "Dash: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, pp. 1–28, 2016.

[15] T. Lee, S. K. Monga, C. Min, and Y. I. Eom, "Memtis: Efficient memory tiering with dynamic page classification and page size determination," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 17–34.

[16] Z. Qi, S. Zheng, Y. Huang, Y. Hui, B. Zhang, L. Huang, and H. Mei, "Chrono: Meticulous hotness measurement and flexible page migration for memory tiering," in *Proceedings of the Twentieth European Conference on Computer Systems*, 2025, pp. 835–853.

[17] W. Tang, Y. Wang, Y. Wang, and J. Wu, "Leave no one behind: Fair and efficient tiered memory management for multi-applications," in *Proceedings of the 54th International Conference on Parallel Processing*, 2025, pp. 699–709.

[18] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang *et al.*, "Tmo: Transparent memory offloading in datacenters," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 609–621.

[19] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[20] R. K. Jain, D.-M. W. Chiu, W. R. Hawe *et al.*, "A quantitative measure of fairness and discrimination," *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, vol. 21, no. 1, 1984.

[21] A. Hall, A. Sarma, E. Choukse, K. Ramachandran, and S. Elnikety, "Harvesting spare cpu resources in container systems," in *NSDI*, May 2026.

[22] Y. Zheng, J. Fan, Q. Fu, Y. Yang, W. Zhang, and A. Quinn, "Agentcgroup: Understanding and controlling os resources of ai agents," *arXiv preprint arXiv:2602.09345*, 2026.

[23] W. Tang, J. Zhu, T. Ai, G. Li, B. Yu, X. Yang, and W. Dou, "Thoth: Provisioning over-committed memory resource with differentiated qos in public clouds," in *Proc. IEEE HPCC/DSS/SmartCity/DependSys*. IEEE, 2023, pp. 82–89.

[24] S. Kumar and V. Kumar, "Karmapm: Reward-driven power manager," in *European Conference on Parallel Processing*. Springer, 2025, pp. 351–365.

[25] H. A. Maruf, Y. Zhong, H. Wang, M. Chowdhury, A. Cidon, and C. Waldspurger, "Memtrade: Marketplace for disaggregated memory clouds," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 7, no. 2, pp. 1–27, 2023.

[26] Compute Express Link Consortium, "Compute express link specification, revision 2.0," https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-2.0-Specification.pdf, 2020, accessed: 2026-07-14.