Dtact vs Tokio Performance Benchmark Analysis Report

This report presents a multi-dimensional benchmark testing and comparative analysis of asynchronous task scheduling performance between **Dtact** and **Tokio** in a specific cloud server environment.

1. Test Environment and Metadata

- **Test Server**: 4-Core Ubuntu Cloud Server
- **Build Commit (Commit Hash)**: [13db4ff883b5675f063edeb3a5269c47f8b2c328]
- **GitHub Workflow Run**: [Workflow Run #26011780070]
- **Benchmarking Tool**: Criterion.rs (default sample size: 200 groups)

2. Raw Data Summary and Comparison

1. Task Spawn Efficiency

This benchmark measures the time taken to spawn and execute a batch of asynchronous tasks at various scales (lower values are better).

Task Scale	Runtime	Lower Bound (Min)	Estimated Mean	Upper Bound (Max)	Outliers	Dtact Speedup
1M ($1,000,00$ **tasks)**	**Dtact**	103.92 ms	104.92 ms	105.94 ms	None	$6.31\times$ **faster**
	Tokio	648.02 ms	662.11 ms	676.42 ms	None	Reference Group
100k ($100,00$ **tasks)**	**Dtact**	11.667 ms	11.807 ms	11.954 ms	6 mild high (3.00%)	$5.34\times$ **faster**
	Tokio	61.064 ms	63.067 ms	65.043 ms	None	Reference Group
10k ($10,00$ **tasks)**	**Dtact**	1.9672 ms	2.0144 ms	2.0620 ms	None	$2.63\times$ **faster**
	Tokio	5.1595 ms	5.2986 ms	5.4395 ms	None	Reference Group

$1k$ ($1,000$ tasks)	**Dtact**	$152.30\ \mu\text{s}$	$157.73\ \mu\text{s}$	$163.31\ \mu\text{s}$	10 mild high (5.00%)	$4.76\times$ **faster**
	Tokio	$719.65\ \mu\text{s}$	$750.89\ \mu\text{s}$	$783.41\ \mu\text{s}$	3 mild high (1.50%)	Reference Group

2. Yield Efficiency

This benchmark measures the time taken for 10 concurrent tasks to perform 100 cooperative yield_now operations each (lower values are better).

Test Case	Runtime	Lower Bound (Min)	Estimated Mean	Upper Bound (Max)	Outliers	Performance Comparison
10 tasks $\times$	**Dtact**	$795.65\ \mu\text{s}$	$827.51\ \mu\text{s}$	$860.19\ \mu\text{s}$	2 mild high, 2 severe high (2.00%)	Approx. $4.41\times$ slower
	Tokio	$179.98\ \mu\text{s}$	$187.73\ \mu\text{s}$	$195.65\ \mu\text{s}$	None	$4.41\times$ **faster**

3. Work Deflection (Hot Core)

This benchmark simulates task dispatching and throttle coordination under highly unbalanced load across a multi-core scheduler. It spans task scales from $1k$ to $10M$ (lower values are better).

Task Scale	Runtime	Lower Bound (Min)	Estimated Mean	Upper Bound (Max)	Outliers	Dtact Speedup
$10M$ ($10,000,000$ tasks)	**Dtact**	$1.6624\ \text{s}$	$1.6792\ \text{s}$	$1.6962\ \text{s}$	2 mild high (1.00%)	$4.13\times$ **faster**
	Tokio[^1]	$6.8482\ \text{s}$	$6.9386\ \text{s}$	$7.0291\ \text{s}$	None	Reference Group
$1M$ ($1,000,000$ tasks)	**Dtact**	$170.21\ \text{ms}$	$172.04\ \text{ms}$	$173.92\ \text{ms}$	5 mild high (2.50%)	$4.00\times$ **faster**
	Tokio	$674.03\ \text{ms}$	$687.77\ \text{ms}$	$701.59\ \text{ms}$	None	Reference

		ms}$	ms}$	ms}$		Group
$100k$ ($100,00$ **tasks)**	**Dtact**	17.472 ms	17.659 ms	17.847 ms	7 mild high (3.50%)	$2.84\times$ **faster**
	Tokio[^2]	49.110 ms	50.114 ms	51.112 ms	2 mild high (1.00%)	Reference Group
$10k$ ($10,00$ **tasks)**	**Dtact**	2.4961 ms	2.5315 ms	2.5675 ms	1 mild low (0.50%)	$2.31\times$ **faster**
	Tokio	5.7240 ms	5.8411 ms	5.9605 ms	2 mild high (1.00%)	Reference Group
$1k$ ($1,00$ **tasks)**	**Dtact**	$273.79\ \mu\text{s}$	$285.23\ \mu\text{s}$	$297.07\ \mu\text{s}$	6 mild high (3.00%)	$2.70\times$ **faster**
	Tokio	$739.64\ \mu\text{s}$	$769.84\ \mu\text{s}$	$801.70\ \mu\text{s}$	7 mild high (3.50%)	Reference Group

[^1]: **Tokio** $10M$ **warning message**: Warning: Unable to complete 200 samples in 600.0s. You may wish to increase target time to 1391.9s, or reduce sample count to 80.

[^2]: **Tokio** $100k$ **warning message**: Warning: Unable to complete 200 samples in 600.0s. You may wish to increase target time to 921.3s, enable flat sampling, or reduce sample count to 110.

3. Deep Data Analysis and Insights

1. Task Spawn Efficiency

- **Absolute Dominance**: Dtact outperforms Tokio by a wide margin in spawning tasks across all scales. At the 1M scale, Dtact takes only 104.92 ms compared to Tokio's 662.11 ms—making Dtact 6.31 **times faster**.
- **Scale Effects**:
  As the volume of concurrent tasks increases, Dtact's performance advantage expands:
  - At 10k tasks, Dtact is $2.63\times$ faster.
  - At 100k tasks, Dtact is $5.34\times$ faster.
  - At 1M tasks, Dtact is $6.31\times$ faster.
    This reveals that Dtact has a lower constant overhead and better concurrent

scalability. For scenarios where a massive number of short-lived tasks must be rapidly spawned, Dtact offers a massive throughput advantage.

2. Coroutine Yield Efficiency

- **Tokio Reclaims Ground**: In the Yield efficiency test, Tokio significantly leads with a mean of $187.73\ \mu\text{s}$ against Dtact's $827.51\ \mu\text{s}$ (approximately $4.41\times$ **faster**).
- **Bottleneck Conjecture**: Tokio leverages a highly optimized Last-In-First-Out (LIFO) slot and a cooperative scheduling budget mechanism, allowing the currently active task to be re-queued instantly on the local queue of the physical thread. Dtact, on the other hand, might be introducing lock contention, memory allocations, or cross-core synchronization overhead during its yield routing, which is further supported by the presence of severe high outliers.

3. Work Deflection (Hot Core) Performance

- **Dtact's Clear Victory**:
  In the Work Deflection test—designed to simulate heavy load imbalances across multiple cores—Dtact maintains a commanding $2.3\times$ **to** $4.1\times$ **speed advantage** over Tokio across all scales:
  - Under the extreme 10M task workload, Dtact smoothly completes the run in only 1.6792 s. Tokio requires 6.9386 s and triggers Criterion's timeout warnings, failing to smoothly collect all 200 samples within the default 600.0 s timeframe (estimating an actual target run time of 1391.9 s).
- **Architectural Analysis**:
  - **Tokio** relies on a classic work-stealing algorithm. When a work imbalance occurs, idle threads must actively steal tasks from the double-ended queues of busy cores (Hot Cores) in a lock-free manner. Under extreme task deflection workloads (like 10M tasks concentrated on a single thread), the constant overhead of stealing attempts combined with cache-line bouncing degrades Tokio's multi-core coordination efficiency.
  - **Dtact** handles this scenario remarkably well. The low latency and complete lack of timeout warnings suggest that Dtact's multi-core task distribution strategy—likely involving a lightweight shared queue or a dynamic steering scheduler design—minimizes task-transfer friction and achieves superior load-balancing efficiency under unbalanced, bursty workloads.