Reducing Per-flow Memory Use in TCP SEARCH

Maryam Ataei Kachooei
Worcester Polytechnic Institute
Worcester, MA, USA
mataeikachooei@wpi.edu

Jae Chung
Viasat
Marlboro, MA, USA
jaewon.chung@viasat.com

Amber Cronin
Akamai
Cambridge, MA, USA
acronin@wpi.edu

Feng Li
Viasat
Marlboro, MA, USA
feng.li@viasat.com

Benjamin Peters
Viasat
Marlboro, MA, USA
benjamin.peters@viasat.com

Mark Claypool
Worcester Polytechnic Institute
Worcester, MA, USA
claypool@cs.wpi.edu

***Abstract*—The Slow start Exit At Right CHokepoint (SEARCH) algorithm is designed to exit the TCP slow start phase after the flow has reached the link capacity but before packets have been lost. To do this, SEARCH keeps a history of the bytes delivered over a recent time window, aggregated into bins. Unfortunately, this delivery history must be kept per-flow, adding additional memory load for each TCP connection. We address this per-flow memory load by observing that SEARCH only needs the relative number of bytes delivered and propose a bit-shifting technique that dynamically compresses bin values as needed. Our approach is tunable to the memory-use reduction required compared to the delivery precision needed. Evaluation of our approach over a satellite network shows SEARCH bin memory use can be reduced by 50% or even 75% without any significant sacrifice in SEARCH algorithm accuracy. Our approach is generalizable to other network algorithms, too, reducing memory use for algorithms that use sliding windows and historical data tracking.**

***Index Terms*—memory use, bit shifting, TCP SEARCH, slow start**

I. Introduction

The Slow start Exit At Right CHokepoint (SEARCH) algorithm [1] is designed to improve TCP's start up by exiting the slow start phase after the congestion point is reached but before packet loss occurs. To do this, SEARCH tracks the delivered bytes over time via a sliding window, comparing the expected delivered bytes to the actual delivered bytes and determining the at-congestion point when there is enough of a difference.

During the slow start phase, TCP increases the congestion window (cwnd) exponentially, doubling the delivery rate each round-trip time (RTT). However, when the network reaches its capacity, the delivery rate stops doubling, indicating TCP should exit from slow start. Figure 1 illustrates this behavior. The left panel shows the alignment of delivered bytes (blue line) and sent bytes (green line) up to the capacity point (dashed vertical blue line). After this point, the sent bytes continue to increase while the delivered bytes plateau. Note, if the sent byte rate continues to increase, the bottleneck queue will fill and eventually induce packet loss.

The right panel highlights the normalized difference between the sent bytes and the delivered bytes. When the normalized difference rises above zero, this indicates that the network's capacity has been reached, typically before packet loss (dashed vertical red line). SEARCH uses this deviation, along with a predefined threshold, to determine the right moment to exit the slow start phase.

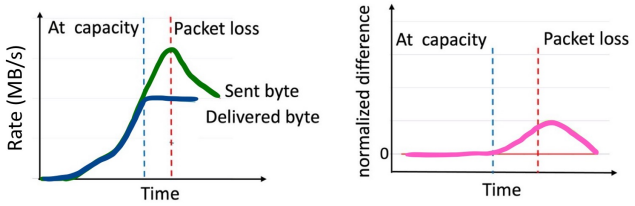

Fig. 1: SEARCH: (left) sent and delivered rates over time, and (right) normalized difference over time.

The SEARCH RFC draft [2] was presented at the Internet Engineering Task Force [3] and singled out for support by their Congestion Control Working Group.[1] As such, critique, deployment and improvements to SEARCH are encouraged. Open source versions of SEARCH are available for Linux[2] and QUIC.[3]

In SEARCH, the sliding window of delivered bytes over time is broken into smaller intervals, called bins, which aggregate delivered byte history over small time periods so as not to store per-packet information. Since SEARCH benefits from fine-grained tracking and must smooth delivery rates in networks with high RTTs, the window requires a large number of bins.[4] Since each bin stores the highest sequence number seen in the delivered acknowledgments, a 32-bit integer (`u32` is used. Unfortunately, the combination of many bins (25) and many bytes (4) for each bin introduces a significant memory overhead, extra storage that must be kept for each TCP flow, limiting scalability, particularly for resource-constrained

[1] https://datatracker.ietf.org/group/ccwg/documents/
[2] https://github.com/Project-Faster/tcp_ss_search
[3] https://github.com/Project-Faster/quicly/tree/generic-slowstart
[4] The number of bins in the SEARCH implementation for Linux is 25.

servers.

To address this challenge, we propose a key optimization – *bit-shifting* the bins to dynamically compress bin values by reducing their precision as needed, significantly lowering the memory footprint of the bins while preserving enough accuracy for the SEARCH algorithm. We incorporate this technique into the TCP SEARCH algorithm, demonstrating how it reduces per-flow memory use while maintaining SEARCH performance. Although demonstrated in SEARCH, this optimization can also be applied to other network protocols that use historical data tracking.

The contributions of this paper are two-fold:

1) We introduce a bit-shifting technique to reduce memory use for algorithms that require historical data tracking.
2) We demonstrate the effectiveness of this technique in the TCP SEARCH algorithm, reducing memory usage per TCP flow while retaining SEARCH functionality.

The remainder of this paper is organized as follows: Section II reviews related work; Section III details the methodology, including the implementation of the bit-shifting technique in the TCP SEARCH algorithm; Section IV evaluates the impact of the technique on memory savings and SEARCH algorithm accuracy; Section V discusses the implications of the research; Section VI mentions some limitations; and Section VII summarizes our conclusions and possible future work.

II. Related Work

Efficient memory management is critical in network protocols that rely on historical data for decision-making. Various techniques have been proposed to reduce the memory overhead associated with tracking past network behavior.

Tangwongsan et al. [4] provide a overview of sliding-window aggregation algorithms, emphasizing their application in data stream processing. While their work explores the computational efficiency of aggregation, it does not address the challenges of reducing memory overhead when maintaining precision across a large amount of historical data.

Oge et al. [5] propose scalable hardware implementations of sliding-window aggregates, focusing on performance optimization. However, their approach requires specialized hardware, making it less practical for software-based congestion control mechanisms like TCP.

Cardwell et al. [6] introduce TCP BBR, a congestion control mechanism that uses estimates of bottleneck bandwidth and RTT to adjust TCP window settings. BBR uses a history of delivered bytes, somewhat similar to that used by SEARCH, so BBR could benefit from the reduced memory afforded by the bit-shifting techniques in our paper.

Kachooei et al. [7] introduce the BEST algorithm to improve the slow start phase by leveraging bandwidth estimates to determine the slow start exit point. BEST uses a window of bandwidth estimations to figure out the link capacity. Since the reliance on a history of bandwidth estimates suffers the same memory burden present in SEARCH, BEST could benefit from the reduced memory afforded by our bit-shifting techniques.

Our work complements these studies by introducing a bit-shifting technique to dynamically compress historical values, significantly reducing memory usage without necessarily compromising accuracy.

III. Methodology

This section describes our approach to reduce per-flow memory use in the TCP SEARCH algorithm. Our technique uses dynamic bit-shifting to compact data as needed, reducing the memory needed for each TCP SEARCH connection.

A. Overview of SEARCH Algorithm

The SEARCH algorithm is designed to exit slow start in a timely manner – after reaching the network's capacity point but before inducing packet loss [2]. SEARCH identifies the congestion point by leveraging the principle that the delivered bytes should approximately double with each RTT until capacity is reached, whereupon the delivered bytes will stay the same each RTT. SEARCH compares the delivered bytes in the current RTT with the expected bytes sent during the previous RTT. When the current delivered bytes is significantly lower than the expected delivered bytes (i.e., it has stopped doubling), that suggests the capacity point has been reached and TCP should exit slow start and enter congestion avoidance. This process of comparing delivered bytes over time requires maintaining a delivery history – the focus of our paper.

While SEARCH has demonstrated its effectiveness in wireless networks [1], to handle fluctuations in RTT inherent in many wireless networks SEARCH maintains a sliding window of delivered bytes spanning multiple RTTs.[5] Rather than tracking individual packet delivery times, the data within the sliding window is aggregated into bins that represent fixed time intervals, significantly reducing memory requirements. The window slides over bin-by-bin rather than at the granularity of each ACK packet, significantly reducing processing overhead, too. SEARCH also holds data from the previous RTT, as well as some extra bins to account for RTT variation.

In total, the latest SEARCH default settings have 10 bins for the current delivered bytes window and 15 extra bins for the previous delivered bytes window, and extra bins to handle RTT variation, resulting in a total of 25 bins. By default, each bin in SEARCH is represented as a 32-bit integer (`u32`) to accommodate the cumulative bytes acked by TCP. This means at a minimum, the 25 bin window needs to be 100 bytes in order to manage the delivered bytes history, storage that needs to be kept per connection by the server. While 100 bytes may not seem like a lot given memory sizes on many devices, the storage here is kernel memory and limits efficiency, scalability, and stability, particularly for memory-constrained systems.

B. Dynamic Bit-Shifting for Compact Data Representation

Figure 2 illustrates how SEARCH fills bins with cumulative delivered bytes over time. The figure depicts an array of bins, each indexed by the integer value at the top. The bins are filled

[5]The window scale factor for Linux is set to 3.5 RTTs.

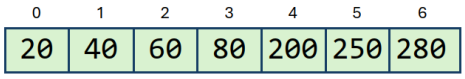

Fig. 2: Bins in SEARCH with default bin size (u32).

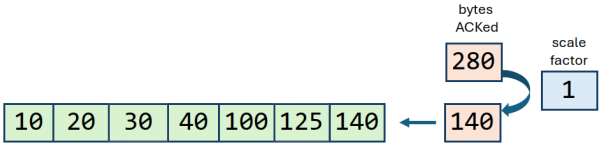

Fig. 3: Bit-shifting applied to u8 bins.

with the cumulative TCP bytes acked values left to right, hence the steady increase in the stored numbers. Under the default configuration where each bin is a u32, all values can fit based on the allotted bits, even the largest value, 280, placed into bin 6. However, if fewer bits were used in each bin, such as u8, for this example, the 280 value would be too large to store in bin 6.

In the case where the acked value is too large to be stored in a bin, the value is scaled by bit-shifting, dividing the value in half and recording that the value was scaled. This repeats (divide by 2, increment the scale factor) until the incoming value can fit into a bin.

Figure 3 illustrates bit-shifting for our current example. When the bin size is a u8, the incoming 280 value cannot fit into a bin. In this case, it is divided by 2 and becomes 140 and the scale factor incremented to 1. Since 140 can fit into a u8 it is added to the array. At this point, all previous bin values are *also* scaled (divided by 2) – i.e., 20 becomes 10, 40 becomes 20 ... and 250 becomes 125.

Shifting all bin values – both the latest incoming ack value and all previously stored ack values – preserves the high order bits and the relative difference in previously acked values and currently acked values. It is this relative difference that SEARCH needs, not the absolute values. Figure 4 depicts the SEARCH window calculations if u32 bins are used (top) compared to u8 bins (bottom). In both cases, the window of delivered bytes is calculated as the difference between the last bin value and the earliest bin value in the window. For this example, with u32 bins:

- Current window = 280 - 60 = 220 bytes
- Previous window = 200 - 20 = 180 bytes

With u8 bins (after bit-shifting):

- Current window = 140 - 30 = 110 bytes
- Previous window = 100 - 10 = 90 bytes

SEARCH computes the normalized difference (norm) of twice the previous delivered bytes compared to the current delivered bytes:

$$\text{norm} = \frac{(2 \times \text{previous\_window}) - \text{current\_window}}{2 \times \text{previous\_window}}$$

Computing the norm for the u32 bins yields 0.375 and for the u8 bins yields 0.375 i.e., the norm is consistent,

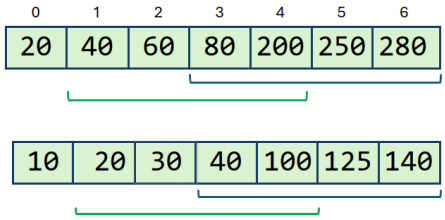

Fig. 4: Window calculations for u32 (top) and u8 (bottom) bin sizes.

demonstrating that SEARCH accuracy in detecting delivered bytes differences is the same in this example even with the bin scaling.

Algorithm 1 shows pseudo-code for the dynamic bit-shifting technique used in SEARCH. The algorithm's inputs in Lines 2-6) assume the maximum value a bin can hold is defined as a constant in *MAX_SIZE* (e.g., with u8 bins, the *MAX_SIZE* would be 255). The bins themselves are an array allocated with *NUM_BINS* (e.g., *NUM_BINS* is 25), indexed at *curr_index* (the latest bin) and scaled with *total_scale* (initially 0). The *acked_bytes* are from the acknowledged TCP sequence numbers in the incoming ack packets.

The bit-shifting algorithm only acts when a bin is being updated – i.e., the current TCP acknowledged byte value needs to be stored in the current bin. Line 9 shifts the acknowledge bytes (*acked_bytes*) by the scale factor (*total_scale*) and stores this in the (local) *bin_value*. On Line 11, if the resulting bin value is larger than the maximum size a bin can hold (*MAX_SIZE*), then Lines 14-17 shift the bin value and increase the current scale factor (*curr_scale*), repeating until the bin value fits into a bin.

Once the necessary current scale factor (*curr_scale*) is determined, Lines 19-21 apply the same number of shifts to all previously stored bins in the window. This ensures that all bin values remain scaled consistently relative to each other.

In Line 23, the total scale factor is updated to the cumulative shifts applied to all past bin values and for subsequent bin values.

Finally, Line 26 stores the scaled bin value in the current bin.

This approach dynamically adjusts bin values only when needed, ensuring minimal overhead in scenarios where bin values remain below the defined threshold (*MAX_SIZE*). When scaling is needed, it is applied to all bin values, preserving the relative values in each bin for comparing previously delivered byte windows with current delivered byte windows, needed by SEARCH.

IV. Evaluation

In this section, we evaluate the performance of the SEARCH algorithm with the proposed bit-shifting technique for memory-use reduction. The evaluation examines how different bin sizes (u32, u16, u8, and u4) impact the accuracy of SEARCH's

Algorithm 1 Dynamic bit shifting for memory optimization.

Input:
$bin[NUM\_BINS]$ // Window of delivered bytes
MAX_SIZE // Maximum allowable bin value
$curr\_index$ // Index of bin to fill
$total\_scale$ // Total shifts (div 2) for all bin values
$acked\_bytes$ // Latest ack sequence number

On each bin update:
// Scale current bin value based on previous shifts
$bin\_value \leftarrow acked\_bytes \gg total\_scale$

// Check if bin value can fit in bin
if $bin\_value > MAX\_SIZE$ **then**
    // Shift bin value until it fits
    $curr\_scale \leftarrow 0$
    while $bin\_value > MAX\_SIZE$ **do**
        $bin\_value \leftarrow bin\_value \gg 1$
        $curr\_scale \leftarrow curr\_scale + 1$
    end while
    // Apply additional shifts to all previous bins
    for $i \leftarrow 0$ **to** $current\_index - 1$ **do**
        $bin[i] \leftarrow bin[i] \gg curr\_scale$
    end for
    // Update total number of shifts
    $total\_scale \leftarrow total\_scale + curr\_scale$
end if
// Store scaled value in current bin
$bin[current\_index] \leftarrow bin\_value$

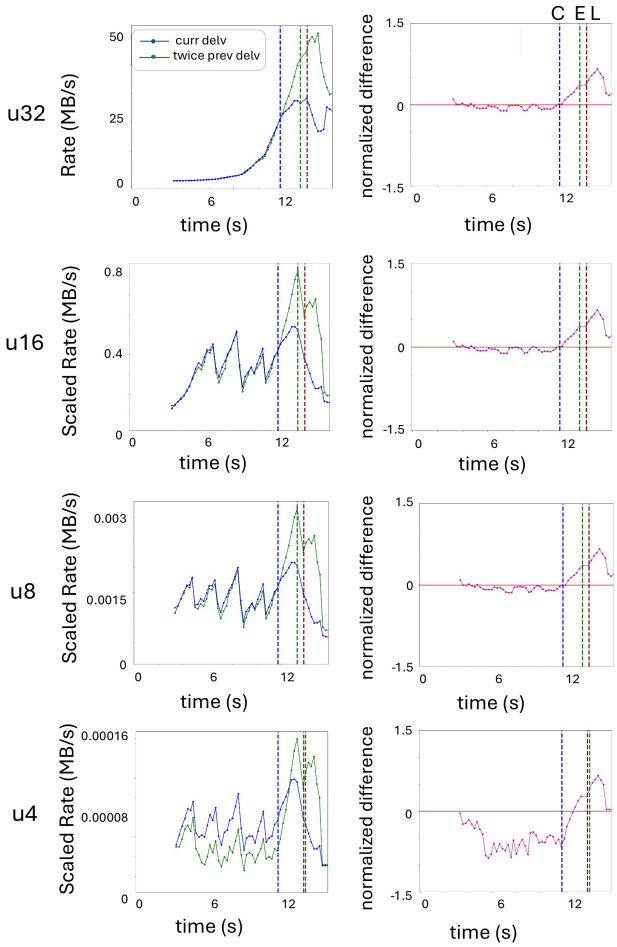

Fig. 5: SEARCH with bit-shifting for bin sizes of `u32`, `u16`, `u8`, and `u4` for one download over a GEO satellite link. Left: Scaled values for current delivered bytes (green) and twice the previously delivered bytes (blue). Right: Normalized difference (pink).

computations, particularly its ability to determine a slow start exit point after the congestion point but before packet loss.

A. Delivered Bytes and Normalized Difference

Figure 5 shows the results of the bit-shifting technique for bin sizes `u32` (default), `u16`, `u8`, and `u4` for one download over a Geostationary satellite network (a Viasat link).

All configurations, including the default `u32`, as well as smaller bin sizes (`u16`, `u8`, and `u4`), are simulated using a Python codebase to replicate the SEARCH algorithm's behavior. The original data is collected with default TCP Cubic – HyStart and SEARCH disabled – with the kernel generating log messages (ack number plus timestamp) for the simulation.

The left side graphs in Figure 5 show the cumulative delivered bytes (blue line) and twice the previously delivered bytes (green line), which SEARCH compares to compute the normalized difference. The right side graphs illustrate the normalized difference (pink line), which serves as the key metric for determining the capacity – when the normalized difference is over a threshold (0.35 by default) SEARCH exits slow start. Vertical dashed lines mark significant events: the blue dashed line (C) indicates when capacity is reached, the green dashed line (E) shows when SEARCH exits the slow start phase, and the red dashed line (L) marks the first packet loss. The top pair of graphs is for the default `u32` bin size, with each pair below having a reduced bin size: `u16`, `u8` and `u4`.

From the figure, comparing the graphs on the left top to bottom shows differences in the delivered bytes patterns – notably, the "zig-zag" shape in the delivered byte lines for reduced bin sizes. Each peak and corresponding trough in these graphs are when the bit-shifting algorithm scales the incoming bytes acked and all previous bins in order to reduce the size of an incoming value to fit the bin size. The normalized values in the graphs on the right show the SEARCH computations of the difference in delivered bytes versus the expected delivered bytes (twice the previous delivered bytes) look almost identical for `u32` and `u16` bin sizes, deviate slightly for the `u8` bin size, and deviate significantly for only the `u4` bin size. In fact, the SEARCH exit points are the same for all bin sizes, except

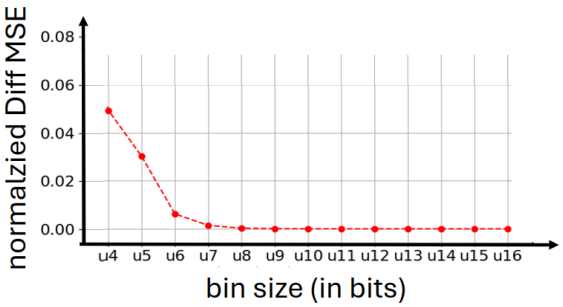

Fig. 6: MSE of normalized difference versus bin size.

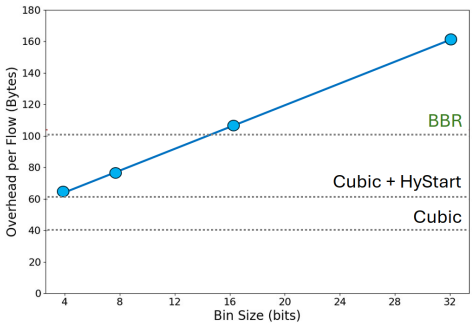

Fig. 7: Per-flow memory overhead versus bin size. Horizontal dashed lines show BBR, Cubic + HyStart, and Cubic for comparison.

for `u4` where SEARCH would have determined the exit point just after packet loss. For this download, the bin size can be safely reduced from `u32` to `u16` or even `u8` without negatively affecting SEARCH's ability to determine the slow start exit point.

To quantitatively compare the impact of reduced bin bits on SEARCH performance, we compute the mean squared error (MSE) between the normalized difference with `u32` bits and the normalized difference with fewer bits. We do so for 45 runs over the same Viasat link as in Figure 5 (which has 1 run). We use a Geo satellite link since prior work has shown these to be among the most challenging for determining link capacity over time [1].

Figure 6 shows the MSE of the normalized difference for different bin sizes. From the figure, bin sizes `u8` and up exhibit minimal error, with the normalized differences used by SEARCH nearly the same as for the `u32` baseline. In contrast, as bit sizes drop to `u7`, `u6`, `u5` and `u4`, the errors increase compromising SEARCH's ability to accurately detect the capacity point — as in Figure 5 bottom, fewer bits makes it more likely SEARCH exits after the capacity point.

The per-flow SEARCH memory use scales linearly with the bit size. Figure 7 illustrates this relationship. The x-axis is the bin size (in bits) and the y-axis is the per-flow memory overhead. The angled blue line shows the per-flow memory used by SEARCH for the given bin bit size. The horizontal lines provide a comparison of the per-flow memory overhead for BBR, Cubic + HyStart, and Cubic, respectively.

Note, while Figure 6 shows analysis of non-standard variable sizes (e.g., 5 bits, 13 bits) this is mostly an academic exercise. Practically, only bin sizes of 8 bits (`u8`) and 16 bits (`u16`) should be considered as alternatives to 32-bit bins. Based on the Figure 6, since `u8` is right at the edge of the MSE trendline, a "safer" choice that has fidelity to the `u32` bin size computations is a `u16` bin size.

Figure 8 compares the per-flow memory overhead for different slow start algorithms and SEARCH. It shows the memory usage for Cubic, Cubic + HyStart, BBR, and Cubic + SEARCH with three different bin sizes (`u8`, `u16` and `u32`). As expected, Cubic has the lowest memory overhead due to its minimal per-flow data tracking (typically exiting slow start only upon packet loss), while SEARCH with larger bin sizes (e.g., `u32`) has significantly more overhead because it stores the full cumulative acked byte values for each bin. However, SEARCH with `u16` and `u8` bin sizes show significant reductions in per-flow memory use – the former having about the same per-flow memory use as BBR and the latter having somewhat less. Note, SEARCH with `u16` bins can fit into Linux's private `ICSK_CA_PRIV` block, which is 104 bytes. This means that `u16` SEARCH can be used in default Linux kernels without requiring kernel modification and recompilation. For resource-constrained devices, smaller bin sizes such as `u8` or below could be considered if memory is at a premium and some inaccuracy in SEARCH is acceptable.

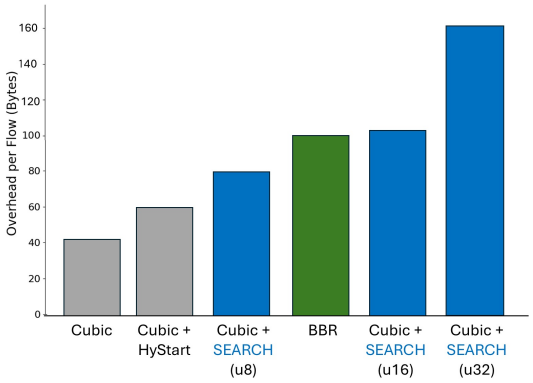

Fig. 8: Per-flow memory overhead. Blue bars are SEARCH with different bin sizes. Green is BBR. Gray bars are Cubic with and without HyStart.

The results presented in Table I highlight the impact of different bin sizes on the performance of TCP with SEARCH based on 45 test cases conducted over the Viasat network. 'Exit' is when slow start exits, 'Early' is the percentage of runs that exit slow start before capacity is reached, 'Late' is the percentage of runs that exit slow start only upon packet loss, 'Chokepoint' is the percentage of runs that exit after capacity is reached but before packet loss. Times are shown in seconds, averaged over all runs with the standard deviation

in parentheses.

From the table, as the bin size decreases, the exit times remain exactly the same for `u16` but show slight variations for other bin sizes. For instance, the average exit time for `u4` is slightly higher at 11.5 seconds compared to 11.3 seconds for `u8` and 11.2 seconds for `u16` and `u32`. The percentage of runs exiting slow start early is consistent at 4.4% for all bin sizes. However, late exits increase from 22.2% for the larger bin sizes to 25% for the `u4` bin size. Correspondingly, the Chokepoint exit percentage decreases from 73.4% for the larger bin sizes down to 70.6% for the `u4` bin size.

TABLE I: SEARCH evaluation with different bin sizes.

Bin Size	Exit (s)	Early (%)	Late (%)	Chokepoint (%)
u4	11.5 (3.0)	4.4	25.0	70.6
u8	11.3 (2.7)	4.4	22.2	73.4
u16	11.2 (2.9)	4.4	22.2	73.4
u32	11.2 (2.9)	4.4	22.2	73.4

Overall, the analysis suggests SEARCH bin sizes can be reduced from `u32` to `u16` with identical algorithm performance, and to `u8` with similar algorithm performance. Decreasing the bin size to `u4` will somewhat degrade algorithm performance.

V. Discussion

The SEARCH algorithm relies upon a history of delivered bytes, so it uses bins to aggregate the delivered bytes over a small time period in order to avoid storing per-ack values. However, the number of bins (25 by default) and the bits required for each (4 bytes per bin) means considerable per-flow overhead (at least 100 bytes for the bins alone).

Based on our analysis, storing the incoming values in a `u32` is not needed – since SEARCH compares previously delivered bytes to currently delivered bytes, the *relative* amounts are all SEARCH really needs. This means fewer bytes – `u16` or even `u8` – can be used for each bin without sacrificing SEARCH accuracy.

Moreover, the approach presented – bit-shifting on demand, only when values get too large – is tunable to different environments. TCP servers that handle only a few connections but are on a high-capacity link may choose to use large bins – `u32` or even larger if the kernel uses larger values for TCP – since per-flow memory overhead is not an issue but fidelity to the acked bytes could be. Conversely, TCP servers on resource-constrained devices may use small bins – `u8` or even smaller – if the per-memory overhead is critical and the network capacity is not large.

When bit-shifting is required – i.e., the incoming value is too large to fit into the bin – there is some CPU overhead in the shift itself and in the shift for each previously-stored bin. There could be multiple shifts required (i.e., the shifting is done in a loop in Lines 14-17), but in practice, there is typically only one shift or at most two.

The bit-shifting approach as presented is targeted for SEARCH, but potentially any algorithm that keeps historical data over time and may be larger than the current storage capability could make use of the bit-shifting technique.

VI. Limitations

The work as presented does not evaluate the processing overhead for bit shifting. While the number of instructions in Algorithm 1 is limited, this could still be significant for some systems.

The analysis of overhead presented only considers the additional memory used by the SEARCH algorithm. There is still per-flow memory used by the base TCP algorithm itself. Analysis of the relative per-flow overhead for SEARCH (with and without smaller bins) could put this in perspective.

The evaluation is for differences in the normalized computation values and effectiveness of SEARCH, but is for only a single dataset on one link type. Additional link types may show different inflection points in MSE analysis (Figure 6) and subsequent SEARCH analysis (Table I).

VII. Conclusion

Efficient memory use is critical for TCP protocols since each connection must maintain per-flow state. TCP protocols that do analysis over time are especially susceptible to higher per-flow overheads since they rely upon stored historical data. TCP with SEARCH is just such a protocol, where historical data is stored in bins and each bin adding to the per-flow overhead for the flow.

To dynamically reduce per-flow memory use while preserving data fidelity for SEARCH effectiveness, we propose a bit-shifting technique that reduces the size of incoming values on the fly. Incoming values are too large to fit are compressed (bit-shifted) along with all previously stored data, and the scale factor stored. This reduces data fidelity but allows relative differences to be compared (e.g., for SEARCH the previous delivered bytes and the current delivered bytes).

Evaluation for SEARCH shows that, compared to the default bin sizes (`u32`), bin sizes that are half as large (`u16`) or even a quarter as large (`u8`) have only minimal error while significantly reducing the per-flow memory overhead. For example, a 16-bit bin size (`u16`) has about the same overhead at TCP BBR and allows TCP with SEARCH to fit into the standard Linux kernel without recompilation.

There are still several areas for future research. Future work could do a more detailed evaluation of SEARCH with different bin sizes, beyond the single (if large) dataset tested in this paper. Other future work could consider the number of extra bins used by SEARCH (15 extra bins by default, where the base is 10 bins) could perhaps be reduced by using a different dynamic-shift technique – here, the bin durations could be increased (e.g., doubled) when the RTT shifts require more extra bins. SEARCH could be combined with other TCP versions (e.g., BBR), perhaps improving upon their default start-up exit conditions. The above work coupled with a broader evaluation of SEARCH – more network link types, bottleneck bandwidth, and round-trip time conditions – could help support and refine SEARCH deployment.

REFERENCES

[1] M. A. Kachooei, J. Chung, F. Li, B. Peters, J. Chung, and M. Claypool, "Improving TCP Slow Start Performance in Wireless Networks with SEARCH," in *WoWMoM Symposium*, 2024, pp. 279–288.

[2] J. Chung, M. A. Kachooei, F. Li, and M. Claypool, "SEARCH – a New Slow Start Algorithm for TCP and QUIC," p. 15, Jul. 2024. [Online]. Available: https://datatracker.ietf.org/doc/draft-chung-ccwg-search/

[3] M. Claypool, "SEARCH – a New Slow Start Algorithm for TCP and QUIC," in *IETF*, Vancouver, Canada, 2024.

[4] K. Tangwongsan, M. Hirzel, and S. Schneider, "Sliding-Window Aggregation Algorithms." 2019.

[5] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie, and T. Yoshinaga, "An Efficient and Scalable Implementation of Sliding-window Aggregate Operator on FPGS," in *Symposium on Computing and Networking*. IEEE, 2013, pp. 112–121.

[6] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," *Communications of the ACM*, vol. 60, no. 2, pp. 58–66, 2017.

[7] M. A. Kachooei, Z. Pinhan, F. Li, J. Chung, and M. Claypool, "Fixing TCP Slow Start for Slow Fat Links," in *0x16 NetDev Conference*, 2022.