# Linked Weak Reference Arrays

## A Hybrid Approach To Efficient Bulk Finalization

Andrés Valloud
LabWare, Inc.
public.andres.valloud@gmail.com

## Abstract

The present work describes a challenging, real-life finalization scenario that applies combined scalability and resource utilization pressure. Neither weak reference arrays nor ephemerons satisfactorily address the performance-critical demands. Confronting these existing limitations requires a new strategy. The proposal is a hybrid weak arrayed container with properties from both weak reference arrays and ephemerons. This approach relies on support from a memory manager allowing dynamic slot reference strength.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic Storage Management

***Keywords*** Smalltalk, GemStone/S, finalization, ephemeron, weak reference, weak array, memory management, garbage collection

## 1. Introduction

This paper discusses Smalltalk memory manager improvements to simultaneously maximize finalization throughput and minimize memory usage. These refinements address scalability limitations exposed by increased resource utilization pressure. Specifically, both weak reference arrays and ephemerons are insufficient when ideal performance is required: weak reference arrays offer compact storage at the cost of comparatively inefficient finalization, while ephemerons provide especially efficient finalization at the cost of increased memory usage. Enhancing a memory manager to support dynamic slot reference strength proved insufficient to reach ideal performance. However, the present proposal builds on these latter enhancements to introduce a new finalization container — the linked weak reference array.

From 2007 to 2014, the author worked on the HPS Smalltalk virtual machine originally described in [1]. GemTalk Systems distributes the GemStone/S Smalltalk implementation, which can be used as an object database. Several GemTalk customers deploy GemStone/S database clients on HPS systems. Over time, strategic teamwork with GemTalk's engineers resulted in significant improvements to HPS' memory manager. This collaboration, now continuing beyond the context of HPS, ultimately led to the present paper.



*IWST '15*, July 15th, 2015, Brescia, Italy.

ACM 978-1-nnnn-nnnn-n/yy/mm.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

## 2. Background

### 2.1 Weak arrays and ephemerons

In general terms, Smalltalk implementations support two modes of object finalization [2]: arrays of weak slots called *weak arrays*, and *ephemerons*.

Briefly, weak array finalization operates in the following manner. When the system's garbage collector detects a weak slot referent is not referenced strongly elsewhere, the corresponding weak array slot is overwritten with a special object called a *tombstone* and the weakly held object is collected. At this point, the weak array is entered in a finalization queue so the application is eventually notified of newly tombstoned slots in the weak array. Upon receiving this notification, the weak array object is said to *mourn*. When a weak array mourns, it carries out finalization actions corresponding to the tombstoned slots.

Note slot tombstoning is carried out by the garbage collector, while mourning is delegated to the application. This responsibility distribution lets system users specify arbitrary finalization actions.

In Smalltalk implementations, the typical tombstone object is an immediate such as the small integer zero. Although some Smalltalk systems use the object `nil` as a tombstone, using immediate objects as tombstones is preferable because it simplifies the semantics of storing non-immediate tombstone objects in weak arrays. That is, since immediate objects are never garbage collected, small integer tombstones cannot be tombstoned themselves.

Recall that Smalltalk indexed slot objects can also have non-indexed fields. Typically, weak arrays have weak indexed slots, and strong non-indexed fields. Thus, weak array mourning requires determining which indexed slots have been tombstoned. Moreover, pragmatic finalization requires copying sufficient information from the weakly held objects (e.g. operating system handles), because weakly held referents will have been collected by the time the weak array mourns. If these copies are not made, there is the risk of referencing the weakly held object strongly. In that case, the weak slot will never be tombstoned.

These weak array inefficiencies were addressed by ephemerons [2]. In contrast with weak arrays, each ephemeron is responsible for finalizing a single object. As a result, ephemeron finalization obviates the need to scan for tombstoned weak slots. An ephemeron's first non-indexed field, called its *key*, is treated as a special case by the garbage collector. For the present discussion, it shall suffice to assume a garbage collector can detect when an ephemeron's key is only referenced by its corresponding ephemeron[1]. When this condition is detected, the garbage collector marks the ephemeron as inactive (by turning it into a regular strongly referencing object), and queues the ephemeron for finalization. Ephemerons queued for

[1] As per [2] and the author's HPS experience, ephemeron semantic edge cases arising from transitive closure considerations can be complex and ambiguous at best.

finalization are said to have been *triggered* — that is, ephemerons are designed to perform finalization once. Critically, triggered ephemeron keys are not tombstoned. Instead, all the information necessary for finalization is still available at the time an ephemeron mourns. Consequently, ephemerons do not need to duplicate data from their keys to perform finalization.

Other systems implement roughly equivalent finalization strategies falling into the preceding two levels of storage granularity [3].

### 2.2 Dynamic reference strength

The GemStone/S system is a full Smalltalk implementation that can also function as an object database [4]. When GemStone/S is used as a repository, other Smalltalk systems deploy client software to transactionally exchange objects with the GemStone/S server. GemStone/S repositories typically grow to hundreds of gigabytes in size. The sheer volume of data demands extremely high performance from the Smalltalk clients. For the purposes of this paper, these performance requirements specifically mandate tight client and server synchronization with regards to which server objects are still referenced by the clients. Consequently, client systems must be able to detect when the application no longer references the local counterparts of objects persisted on the GemStone/S server.

An obvious approach is to let finalization drive the client server synchronization. Due to object volume, using ephemerons for this purpose is impractical. Each ephemeron would impose significant memory overhead for each and every persisted object: an object header for the ephemeron, and at least one field for the ephemeron to reference the persisted object. Since most objects are comparatively small[2], the ephemeron induced overhead is significant relative to the underlying persistent objects' size. In addition, the memory manager system on the client side would have to be engineered to handle potentially millions of triggered ephemerons at a time. These amounts are well outside the design envelope for ephemerons. Moreover, triggering millions of identical finalization actions individually is inefficient. Consequently, GemStone/S clients hold on to persisted objects using weak arrays.

With this configuration, a performance issue arises because automatic garbage collection may untimely place tombstones in the GemStone/S client's weak arrays. Concretely, object transaction processing is far more efficient when the weak arrays can be assumed to have no tombstones for the duration of the transaction. However, the client system may be forced to address a low memory condition by invoking the garbage collector at any time. While coping strategies exist, eventually the alternative to unwanted tombstones will be a system crash. Thus, entirely disabling the garbage collector to prevent tombstones is unfeasible.

At first glance, it appears strong copies of the client's weak arrays must be made during transaction processing to keep the weak arrays tombstone-free. This copying can require significant time, create voluminous amounts of garbage, and might be impossible due to lack of memory. In short, copying the weak arrays into strong arrays is generally incompatible with reliability and high transaction throughput. The alternative, tolerating weak array tombstones at arbitrary times, is just as unsatisfactory from a performance perspective.

To address this problem, the author extended and improved the HPS memory manager to allow dynamically adjusting a weak array's weak slot reference strength. In other words, applications were given the ability to make a weak array's weak slots start behaving as strong references. Just as importantly, previously weak and now strong arrays could be made weak again. This functionality was implemented in general fashion, allowing changes in weakness for weak arrays, changes in ephemerality for ephemerons, dynamic object class changes, as well as one-way and two-way Smalltalk become operations[3]. Most of the necessary changes involved HPS' incremental garbage collector, because any of these dynamic operations can easily invalidate the collector's incremental mark stack, incremental weak list, and incremental ephemeron list. These enhancements eliminated the need to copy the GemStone/S client's weak arrays into strong arrays. Instead, the client software simply turns weak arrays into strong arrays during the times when the appearance of tombstones is undesirable. This approach has been in production for a number of years [9], and works without apparent problems to the author's knowledge [7].

Nevertheless, once weak array copying was made unnecessary, a new issue became apparent. Weak array mourning still requires scanning to find tombstoned slots, and the persistent object tables can grow very large. Mourning became a significant time consuming operation preventing higher transaction throughput. This paper's proposal is designed to address this specific scenario.

## 3. Problem statement

To summarize, the goal is to minimize the time needed to detect local persistent object counterparts no longer referenced on the client side of a GemStone/S client server interaction. Such objects are already detected by, and known to, the garbage collector. But, since ephemerons are unsuitable for this application due to prohibitive memory and processing overhead, the garbage collector has no other efficient way to communicate that knowledge to the application. As a result, there is significant inefficiency measured in terms of linearly scanning for tombstones during weak array mourning.

Mitigation strategies might offer some relief, but are generally insufficient. Consider for example breaking up the weak array tables into smaller table pages. This scheme might reduce the time spent scanning for tombstones in exchange for modest memory overhead due to the pages' object headers. Unfortunately, accurately predicting the weak table tombstoning pattern is impossible and/or impractical. Consequently, the added table page complexity will only be amortized when the tombstoning rate is low. Nevertheless, even with a low tombstoning rate, the resulting table page mourning rate may be high especially when the weak persistent object tables are large. Faster linear tombstone scanning (perhaps using primitives) may offer limited relief, yet the underlying algorithm is still linear search. Finally, in a typical generational scavenger scheme, weak arrays could be queued for finalization with a frequency proportional to that of scavenges.

These limitations suggest a different finalization mechanism is needed.

## 4. Proposed approach

The critical observation is that the typical weak array mourning implementation is inherently inefficient. Scanning for tombstones will reconstruct information that was available during garbage collection. The root cause of this inefficiency is that the garbage collector does not record which weak array slots were tombstoned. Hence, the proposal is that small integers other than zero are used to tombstone weak array slots. Specifically, these tombstones shall encode a linked list of tombstoned weak slot indexes terminated with the index zero. The linked list's first index shall be recorded in the first non-indexed field of the weak array. These special types of weak arrays will be called *linked weak arrays*.

[2] This phenomenon is generally due to arguments analogous to those relying on Zipf's Law.

[3] A one-way become replaces all references to object $a$ with references to object $b$. A two-way become exchanges references to $a$ and $b$. Object class pointers are usually unmodified by become operations.

For example, suppose a linked weak array with six indexed slots is tombstoned at indexes 2 and 5. After the garbage collector runs, the linked weak array shall be left as follows (recall Smalltalk indexed slots are 1-based):

| 2 | other non-indexed fields | ? | 5 | ? | ? | 0 | ? |
|---|---|---|---|---|---|---|---|

The tombstone linked list evidently eliminates the need to scan for tombstones during mourning. Moreover, it is hard to envision a more compact encoding. Enhancing a garbage collector to tombstone weak slots this way is trivial. Typical C memory manager implementations would need to change a loop such as

```
idxSlotArray = idxSlotsPtr(weakArray);
for (k = 0; k < numIdxSlots(weakArray); k++)
  if (!isMarked(idxSlotArray[k]))
    idxSlotArray[k] = asSmallInt(0);
```

to the following:

```
nextLinkedListWrite = fieldsPtr(weakArray);
idxSlotArray = idxSlotsPtr(weakArray);
for (k = 0; k < numIdxSlots(weakArray); k++)
  if (!isMarked(idxSlotArray[k])) {
    *nextLinkedWrite = asSmallInt(k);
    nextLinkedWrite = idxSlotArray + k;
  }
*nextLinkedWrite = asSmallInt(0);
```

Modifications along the lines of the above illustration ought to have negligible impact on a memory manager's performance.

So far, linked weak arrays have a lot in common with traditional weak arrays. Nonetheless, in some memory manager systems, the garbage collector may be invoked between the time a weak array is placed in the finalization queue and the time the weak array is done mourning[4]. Depending on the code executed within this time window, additional slots may be tombstoned in the weak array. This condition is not exceedingly problematic for traditional weak arrays. However, the tombstone linked list encoding is vulnerable to race conditions. Explicitly preventing all variants of this phenomenon with e.g. mutexes is deemed too onerous in practice. Therefore, it appears linked weak arrays queued for finalization must mourn between every single garbage collection tombstoning pass.

Fortunately, there is an elegant way to address these data races. When the garbage collector triggers an ephemeron and places it in the finalization queue, it also turns the ephemeron into a regular strongly referencing object. Linked weak array race conditions can be avoided by adopting the ephemeron finalization convention. When the garbage collector places a linked weak array in the finalization queue, it will also change the linked weak array's object header flags to match those of a regular strong array. The now strong array can be made a linked weak array again by the mourning procedure, after tombstone processing is complete. In this way, linked weak arrays can be seen as a hybrid between existing weak arrays and ephemerons. This approach requires dynamic reference strength management facilities, such as those implemented for HPS and described in the previous section [9].

A flexible, robust implementation should dedicate sufficient object header garbage collection bits to differentiate linked weak array semantics from that of regular strong objects, traditional weak arrays, and ephemerons. These header bits enable the dynamic object strength change operations on any suitable object, not just the instances of a few special classes. In addition, clearly distinguishing semantics in this way enables existing application code to continue working as expected without modification.

Although the garbage collector may tombstone weak indexed slots with small integers, nothing in this proposal prevents a linked weak array from storing small integers. The tombstone linked list begins at a non-indexed field, so there can be no confusion between newly tombstoned slots and already present small integers.

It might be argued that the tombstone linked list exposes internal memory manager implementation details. However, linked weak arrays do not break encapsulation any more than weak arrays and ephemerons already do. Specifically, the tombstone linked list could be derived for regular weak arrays during mourning, using no additional information from the memory manager. Simply put, this proposal's argument is strictly one of convenience and efficiency.

## 5. Performance measurements

Linked weak arrays have not been implemented by the author at the time of this writing. Notwithstanding, the limited nature of the required memory manager changes greatly facilitates verifying the expected mourning performance improvement due to linked weak arrays. For example, in Cuis Smalltalk [5] running on Cog [6], it is simple to construct a linked weak array simulation. The class `WeakArray` might implement these two methods:

```
WeakArray>>mourn
  1 to: self size do:
    [:each |
      (self at: each) == 0
        ifTrue: [self mournAt: each]
    ]

WeakArray>>mournAt: anIndex
  "Token least possible mourning cost"
  ^self
```

The `WeakArray` subclass `LinkedWeakArray` might refine `mourn` as shown below. In the code, `firstTombstoneIndex` is an accessor for the first `LinkedWeakArray` instance variable.

```
LinkedWeakArray>>mourn
  | nextIndex |
  nextIndex := self firstTombstoneIndex.
  [nextIndex == 0] whileFalse:
    [
      self mournAt: nextIndex.
      nextIndex := self at: nextIndex
    ]
```

The measurements compare `mourn` execution time spent in a variety of equivalent regular and linked weak arrays. The different array sizes and tombstone rates for the test were chosen as follows:

```
sizes := OrderedCollection with: 1.
8 timesRepeat: [sizes add: sizes last * 2].
6 timesRepeat: [sizes add: sizes last * 4].
rates := OrderedCollection with: 1.
[rates last > sizes last] whileFalse:
  [
    lastCount := sizes last // rates last.
    newRate := rates last.
    [sizes last // newRate = lastCount]
      whileTrue:
        [newRate := newRate + 1 * 5 // 4].
    rates add: newRate
  ].
rates removeLast
```

[4] For instance, responding to a critically low memory condition should preempt finalization, and addressing a memory emergency typically requires invoking the garbage collector.

The work of the memory manager was simulated thus:

```
WeakArray>>tombstoneEvery: aRate
  | thisIndex maxIndex |
  self atAllPut: nil.
  thisIndex := 1.
  maxIndex := self index + 1.
  [thisIndex < maxIndex] whileTrue:
    [
      self at: thisIndex truncated put: 0.
      thisIndex := thisIndex + aRate
    ]

LinkedWeakArray>>tombstoneEvery: aRate
  | thisIndex maxIndex lastLinkIndex |
  self atAllPut: nil.
  lastLinkIndex := 0.
  thisIndex := 1.
  maxIndex := self index + 1.
  [thisIndex < maxIndex] whileTrue:
    [
      | writeIndex |
      writeIndex := thisIndex truncated.
      self
        tombstoneAt: lastLinkIndex
        put: writeIndex.
      lastLinkIndex := writeIndex.
      thisIndex := thisIndex + aRate
    ].
  self tombstoneAt: lastLinkIndex put: 0

LinkedWeakArray>>
  tombstoneAt: anIndex
  put: link
    anIndex = 0
      ifTrue: [self firstTombstoneIndex: link]
      ifFalse: [self at: anIndex put: link]
```

Note higher tombstone rates indicate less slots are tombstoned, and that a tombstone rate of 1 implies every slot is tombstoned.

For each array size and tombstone rate, equivalent regular and linked weak arrays were prepared using `tombstoneEvery:`. Then, sufficient `mourn` iterations were performed so that each measurement loop took at least one second. Finally, a quotient between the average `mourn` execution time for regular and linked weak arrays was derived. If this quotient is exactly 1, this means that regular and linked weak array mourning run just as quickly. If the quotient is greater than 1, linked weak array mourning is faster by a factor equal to the quotient. That is, if the quotient is 3, this means linked weak array mourning was three times faster than regular weak array mourning given a certain array size and tombstone rate. In this context, saying something is $x$ times faster means it will complete $x$ times as many iterations within the same time. This quotient was calculated for all pairs of array sizes and tombstone rates.

Generally, linked weak arrays mourned faster than regular weak arrays in linear proportion to the array size and tombstone rate. Surprisingly, linked weak array mourning was at least $30\%$ faster in *every* case, even for arrays of size 1 and for arrays tombstoned at every index. The entire quotient result set is shown in the appendix.

## 6. Additional benefits, further work

The information written by the garbage collector can be used to achieve further improvements. For example, tombstoned weak array slots are usually recycled by the application. Without linked weak arrays, finding tombstoned slots to reuse requires work. With linked weak arrays, however, this housekeeping is simplified. During mourning, the linked list written by the garbage collector can be easily appended to a free slot list (or list of free slot lists) rooted in a suitable non-indexable field. This arrangement avoids the need to linearly scan for a tombstoned slot in the presence of different tombstone objects. Weak array non-indexed fields' reference strength is typically strong, making them well suited to holding on to structures that must persist across tombstoning events.

Consider what happens when the garbage collector is invoked after a traditional weak array is placed in the finalization queue. It might occur that additional weak array weak slots are tombstoned by this subsequent garbage collector pass. By the time the weak array mourns, whether the tombstones were placed by one or several garbage collection passes will be irrelevant for the purposes of mourning. But what if the second garbage collection pass takes place while the weak array is mourning? The unexpected additional tombstones could introduce data races in the mourning process, and accounting for this possibility makes robust mourning implementations more difficult to construct. In practice, weak array mourning is implemented to tolerate so-called *double mourning*, potentially at great computational cost. However, linked weak arrays do not suffer from these problems. Linked weak array referents will be referenced strongly until mourning determines it is safe to make the linked weak array slots weak again. Thus, whether traditional weak arrays should also adopt the ephemeron finalization convention merits consideration.

Although the present work was specifically designed to improve GemStone/S performance, the proposed solution is applicable to other problem domains. For instance, efficient Smalltalk symbol tables are typically implemented in terms of hash bucketed sets, where each hash bucket is a weak array. Interning new symbols requires finding available indexed slots in a hash bucket, that is, scanning for tombstones. An efficient symbol table implemented with linked weak array hash buckets would enable higher symbol internment throughput.

## 7. Related work

Although VA Smalltalk [8] tombstones weak array slots with `nil`, its finalization queue stores triplets consisting of a weak array instance, the slot index at which said weak array was tombstoned, and the slot's contents prior to tombstoning. Note how the finalization queue introduces a strong reference to previously weakly held objects that would have been collected otherwise. This is unlike other Smalltalk weak array implementations, as well as linked weak arrays, in which tombstones indicate the previously weakly held objects have been garbage collected[5]. Instead, `EsWeakArray` implements behavior characteristic of ephemerons.

The finalization approach of VA Smalltalk prevents scanning for tombstones at the expense of significantly higher memory utilization. Under pressure, a finalization queue up to three times as large as all `EsWeakArray` instances combined may be required. If the finalization queue overflows, thorough system finalization requires multiple garbage collection passes. Even scanning for tombstones would be almost certainly preferable to additional garbage collection activity. Bulk `EsWeakArray` finalization incurs comparatively larger overhead because tombstoned slots are handled one at a time. Arguably, in high volume finalization scenarios, there will be numerous finalization triplets for a given weak array instance. In those cases, the triplets' redundant encoding requires a finalization queue much larger than necessary.

In VA Smalltalk, linked weak arrays would be more efficient at handling combined memory and performance pressure scenarios

[5] Tombstoned data that has also been collected cannot be *resurrected* e.g.: by sending `allInstances`.

where `EsWeakArray`'s ephemeron-like semantics are unneeded. At worst, the finalization queue would only need to hold linked weak array instances, as opposed to triplets for every weak array slot. Hence, finalization queue overflows and multiple garbage collection passes would become a mostly theoretical concern. In contrast with VA Smalltalk's finalization queue triplets, linked weak arrays would need just a single non-indexed field per array. This fixed cost can be amortized easily by increasing array sizes. Linked weak arrays' more compact representation would lessen memory pressure, presumably reducing garbage collection activity. Bulk linked weak array finalization has minimal processing overhead in part because, unlike VA Smalltalk finalization, each linked weak array processes all its tombstoned slots as a group. For example, VA Smalltalk finalization requires at least three `at:` sends per tombstoned slot, while linked weak array finalization would only need one `at:` send per tombstoned slot. Finally, while a linked list of tombstoned slots could be constructed with moderate effort using VA Smalltalk's finalization queue triplets, linked weak arrays would provide that data structure at virtually zero cost.

## 8. Conclusions

Weak arrays and ephemerons are insufficient to address certain usage scenarios that simultaneously demand memory and execution efficiency. Specifically, when memory pressure forces the use of regular weak arrays, scanning for tombstoned slots can become an unavoidable time sink. The root cause of this inefficiency is that traditional garbage collector design discards critical information, forcing mourning to reconstruct this data. This paper proposes that the garbage collector encodes a linked list of tombstoned slots in objects with special garbage collection semantics called linked weak arrays. Linked weak arrays are a hybrid between weak arrays and ephemerons. With this approach, memory efficiency is preserved and tombstone scanning is eliminated. Implementing this functionality without introducing data races is greatly facilitated by a memory manager with dynamic reference strength management capabilities, such as the one implemented in HPS. In addition, in the author's opinion, enough object header bits should be reserved to identify the garbage collector semantics required for each object: strong object, (untriggered) ephemeron, weak array, and linked weak array. Two bits should suffice for this purpose. Other memory management race problems such as double mourning can be addressed by the same mechanism. Finally, it should be noted even the most trivial weak array mourning cases perform better with linked weak arrays than with traditional weak arrays.

## Acknowledgments

The author wishes to thank Martin McClure of GemTalk Systems for years of productive, positive collaboration.

## References

[1] L. Peter Deutsch, Allan M. Schiffman. 1984. Efficient implementation of the Smalltalk-80 system. In POPL '84 Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ACM, New York, NY, USA, 297–302. DOI=10.1145/800017.800542, http://dx.doi.org/10.1145/800017.800542.

[2] B. Hayes. 1997. Ephemerons: a new finalization mechanism. In Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '97), A. Michael Berman (Ed.). ACM, New York, NY, USA, 176–183. DOI=10.1145/263698.263733, http://doi.acm.org/10.1145/263698.263733.

[3] R. Jones, A. Hosking, E. Moss. 2012. The Garbage Collection Handbook. CRC Press. ISBN: 978-1-4200-8279-1.

[4] GemTalk Systems, Inc. http://www.gemtalksystems.com.

[5] Cuis Smalltalk, by Juan Vuletich et al. http://www.jvuletich.org/Cuis.

[6] Cog VM, by Eliot Miranda et al, version 4.0.3164. http://www.mirandabanda.org.

[7] Martin McClure, personal communication.

[8] Instantiations, Inc. http://www.instantiations.com/products/vasmalltalk.

[9] A. Valloud. Object Memory Management, presented at the Smalltalks 2010 conference with subsequent updates at the ESUG 2011 and STIC 2012 conferences. See http://www.fast.org.ar, http://www.esug.org, and http://www.stic.org.

## A. Performance measurement results

The below table lists all performance quotients between linked and regular weak array mourning times for every array size and tombstone rate pair. Quotients greater than 1 indicate linked weak array mourning is faster than traditional weak array mourning. Each column corresponds to an array size, and each row shows quotients for a particular rate. Some pair quotients are blank because tombstone rates higher than the array size do not provide additional information — recall a rate of e.g. 10 means that one in every ten slots is tombstoned. As indicated earlier, these figures compare the efficiency of the least required mourning work. These tests were performed on the Cog VM [6], running on an Intel i7-4850HQ CPU at a base frequency of 2.3 GHz.

| Rate / Size | $2^0$ | $2^1$ | $2^2$ | $2^3$ | $2^4$ | $2^5$ | $2^6$ | $2^7$ | $2^8$ | $2^{10}$ | $2^{12}$ | $2^{14}$ | $2^{16}$ | $2^{18}$ | $2^{20}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1.33 | 1.35 | 1.32 | 1.33 | 1.26 | 1.30 | 1.29 | 1.29 | 1.30 | 1.31 | 1.29 | 1.30 | 1.29 | 1.27 | 1.29 |
| 2 | | 1.75 | 1.96 | 2.14 | 2.42 | 2.20 | 2.29 | 2.34 | 2.37 | 2.38 | 2.39 | 2.39 | 2.37 | 2.36 | 2.34 |
| 3 | | | 1.89 | 2.53 | 2.99 | 3.23 | 3.05 | 3.19 | 3.33 | 3.40 | 3.38 | 3.38 | 3.39 | 3.34 | 3.38 |
| 5 | | | | 3.24 | 3.93 | 4.63 | 4.27 | 4.98 | 5.19 | 5.42 | 5.58 | 5.56 | 5.45 | 5.36 | 5.45 |
| 7 | | | | 3.25 | 4.45 | 6.00 | 6.38 | 6.44 | 7.03 | 7.43 | 7.51 | 7.66 | 7.57 | 7.35 | 7.36 |
| 10 | | | | | 5.79 | 7.17 | 8.63 | 8.21 | 9.59 | 10.37 | 10.55 | 10.69 | 10.55 | 10.40 | 10.34 |
| 13 | | | | | 6.31 | 9.33 | 12.21 | 13.46 | 13.35 | 14.94 | 15.15 | 15.49 | 15.49 | 14.98 | 15.17 |
| 17 | | | | | | 12.12 | 14.64 | 16.04 | 15.80 | 18.68 | 19.61 | 19.71 | 19.54 | 19.37 | 19.40 |
| 22 | | | | | | 11.43 | 17.12 | 19.99 | 19.75 | 23.48 | 24.45 | 24.73 | 24.57 | 24.02 | 23.73 |
| 28 | | | | | | 11.36 | 17.07 | 22.89 | 25.38 | 28.56 | 30.80 | 30.63 | 30.51 | 29.78 | 29.04 |
| 36 | | | | | | | 21.92 | 26.83 | 30.90 | 34.85 | 38.62 | 39.33 | 39.02 | 29.01 | 33.05 |
| 46 | | | | | | | 22.01 | 32.62 | 38.09 | 42.75 | 47.88 | 49.65 | 48.98 | 31.76 | 36.64 |
| 58 | | | | | | | 21.66 | 32.41 | 43.68 | 52.81 | 59.18 | 61.39 | 61.18 | 39.24 | 41.71 |
| 73 | | | | | | | | 42.25 | 51.65 | 60.03 | 72.95 | 76.75 | 76.55 | 55.01 | 54.52 |
| 92 | | | | | | | | 42.17 | 63.09 | 73.38 | 91.03 | 95.58 | 95.26 | 89.13 | 59.37 |
| 116 | | | | | | | | 42.43 | 63.23 | 105.15 | 110.47 | 118.52 | 122.46 | 111.15 | 68.52 |
| 146 | | | | | | | | | 83.46 | 114.89 | 135.93 | 148.47 | 150.20 | 142.95 | 103.49 |
| 183 | | | | | | | | | 82.53 | 147.00 | 166.60 | 185.74 | 183.60 | 185.64 | 114.37 |
| 230 | | | | | | | | | 82.62 | 170.53 | 202.78 | 230.33 | 324.01 | 234.85 | 155.12 |
| 288 | | | | | | | | | | 202.27 | 235.29 | 282.41 | 291.46 | 283.17 | 171.90 |
| 361 | | | | | | | | | | 244.53 | 293.27 | 346.58 | 367.79 | 338.45 | 243.45 |
| 452 | | | | | | | | | | 248.57 | 377.57 | 429.68 | 453.43 | 421.43 | 366.35 |
| 566 | | | | | | | | | | 331.35 | 458.59 | 535.39 | 556.52 | 563.34 | 484.58 |
| 708 | | | | | | | | | | 325.71 | 579.33 | 623.43 | 692.63 | 707.53 | 604.44 |
| 886 | | | | | | | | | | 324.45 | 672.51 | 764.24 | 869.92 | 883.08 | 682.49 |
| 1108 | | | | | | | | | | | 798.45 | 923.78 | 1057.44 | 1098.15 | 855.86 |
| 1386 | | | | | | | | | | | 980.43 | 1147.28 | 1303.74 | 1357.69 | 1074.08 |
| 1733 | | | | | | | | | | | 984.20 | 1497.25 | 1627.28 | 1447.08 | 1357.68 |
| 2167 | | | | | | | | | | | 1307.36 | 1816.96 | 1979.58 | 2114.06 | 1993.21 |
| 2710 | | | | | | | | | | | 1314.82 | 2039.99 | 2331.37 | 2611.88 | 2723.40 |
| 3388 | | | | | | | | | | | 1315.70 | 2664.39 | 2945.72 | 3225.89 | 3326.48 |
| 4236 | | | | | | | | | | | | 3178.02 | 3485.84 | 4027.84 | 4177.18 |
| 5296 | | | | | | | | | | | | 3173.84 | 3907.71 | 4979.14 | 5200.80 |
| 6621 | | | | | | | | | | | | 4031.03 | 5931.98 | 6139.07 | 6405.03 |
| 8277 | | | | | | | | | | | | 5166.64 | 7226.23 | 7487.76 | 8107.55 |
| 10347 | | | | | | | | | | | | 5163.30 | 8102.75 | 9029.40 | 9913.28 |
| 12935 | | | | | | | | | | | | 5204.57 | 9134.19 | 11043.90 | 12288.00 |
| 16170 | | | | | | | | | | | | 5163.47 | 10699.07 | 13136.72 | 15396.71 |
| 20213 | | | | | | | | | | | | | 12628.30 | 15522.34 | 18901.09 |
| 25267 | | | | | | | | | | | | | 15707.99 | 21505.62 | 23183.31 |
| 31585 | | | | | | | | | | | | | 15601.48 | 25675.80 | 27986.01 |
| 39482 | | | | | | | | | | | | | 20646.77 | 31698.65 | 34897.23 |
| 49353 | | | | | | | | | | | | | 20821.33 | 35986.80 | 41275.72 |
| 61692 | | | | | | | | | | | | | 20694.73 | 41735.20 | 52252.24 |
| 77116 | | | | | | | | | | | | | | 49551.61 | 60535.85 |
| 96396 | | | | | | | | | | | | | | 61055.33 | 86784.40 |
| 120496 | | | | | | | | | | | | | | 61344.46 | 102330.31 |
| 150621 | | | | | | | | | | | | | | 80609.93 | 124014.28 |
| 188277 | | | | | | | | | | | | | | 81332.53 | 144179.20 |
| 235347 | | | | | | | | | | | | | | 81314.40 | 164802.79 |
| 294185 | | | | | | | | | | | | | | | 200226.55 |
| 367732 | | | | | | | | | | | | | | | 244799.13 |
| 574583 | | | | | | | | | | | | | | | 324609.02 |