Eliminating Cycles in Weak Tables

Alexandra Barros
(Pontifical Catholic University of Rio de Janeiro
alexandra.barros@gmail.com)

Roberto Ierusalimschy
(Pontifical Catholic University of Rio de Janeiro
roberto@inf.puc-rio.br)

Abstract: Weak References constitute an elegant mechanism for an application to interact with its garbage collector. In most of its typical uses, weak references are used through weak tables (e.g., Java's `WeakHashMap`). However, most implementations of weak tables have a severe limitation: Cyclic references between keys and values in weak tables prevent the elements inside a cycle from being collected, even if they are no longer reachable from outside. This ends up bringing difficulties to the use of weak tables in some kinds of applications.

In this work, we present our approach for overcoming this problem in the context of the Lua programming language. Our approach consists of an adaptation of the *ephemerons* mechanism to tables. We modified the garbage collector of the Lua virtual machine in order to offer support to this mechanism. With this adapted garbage collector we could verify the efficiency and effectiveness of the implementation in solving the problem of cycles on weak tables in Lua.

Key Words: Garbage collection, weak tables, weak references
Category: D.3.3

1 Introduction

Programming languages with automatic memory management usually offer an interface between the *client program* and the garbage collector. This interface consists of mechanisms that allow the client program to interact with the garbage collector and is typically represented by *finalizers* [Atkins and Nackman 1988, Boehm 2003, Dybvig et al. 1993, Hayes 1992] and *weak references* [Leal 2005].

A finalizer is a special method executed automatically by the garbage collector before recycling the memory occupied by an object. Such methods are used in many activities, including managing of object caches and releasing resources provided by servers and other programs. Finalizers have an asynchronous nature. Bloch [Bloch 2001] says that finalizers are unpredictable, frequently dangerous and unnecessary, and have a negative impact on the programs performance. However, Boehm [Boehm 2003] argues that the use of finalizers is essential in some cases and its asynchronism does not necessarily leads to unsound programs.

A weak reference is a reference that doesn't protect an object from being collected by the garbage collector. Many programming languages with garbage col-

lection, at least since the 80s, offer some support to weak references [PARC 1985, Rees et al. 1984]. Amongst other applications, weak references can be used as a finalization mechanism, avoiding many problems associated with traditional finalizers. Depending on the programing language and on the way the garbage collection management is done, the support to finalizers may even become unnecessary.

The increasing importance of weak references motivated our search for a solution for one critical problem: cycles in weak tables. A typical example of this problem occurs with property tables. Frequently, we want to add properties to an object dynamically and independently from its class attributes. A very common approach is to use a *property table*. In a property table, a reference to an object is inserted as the search key, and the value associated to this key specifies extra properties. However, if all references in the property table were ordinary, the simple fact of inserting a new key/value pair would prevent the collection of the object referred by the key. The best way to solve this problem is with *weak tables*, which is a data structure implemented via weak references. A weak table comprises *weak pairs* where the first element, the key, is maintained by a weak reference and the second element, the value, is maintained by an ordinary reference. This way, adding a property to an object does not prevent its collection.

However, the problem with cycles in weak tables still occurs in most programming languages: the existence of cyclic references between keys and values prevents the elements in the cycle from being collected, even if the client program does not have other references to them. A frequent scenario for these cycles consists of a value that refers to its own key. For example, a property table (implemented using a weak table) may associate functions to their respective modules, so that it can say to which module a given function belongs to. As each module refers back to all its functions, no key or value in this property table will ever be collected. This ends up causing a relevant waste of memory and imposes difficulties in the use of weak tables. This problem can be found, for example, in programming languages like Java[SUN 2006] and Lua [Ierusalimschy 2006]. Through Lua's discussion list we could observe frequent complaints related to this problem.

A solution to this problem can be found in a mechanism called *emphemerons*, presented by Hayes [Hayes 1997]. Jones [Jones et al. 1999] independently developed a similar solution for the Haskell [Glasgow 2007] programming language, containing the same core idea as the ephemerons mechanism.

Based on the success obtained in Haskell, we designed and implemented an adaptation of this mechanism for Lua. As a starting point, we studied carefully the algorithm described in the work of Hayes [Hayes 1997]. As we studied this mechanism we were curious about why it is not implemented in most pro-

gramming languages and what is the impact of implementing it. Afterwards, we developed an adaptation for the Lua garbage collector in order to measure this impact. Our aim is to show that the problem of cycles in weak tables can be easily solved for Lua by the incorporation of ephemerons. And by doing so, we are able to improve the weak reference mechanism.

2 Weak Tables

Weak tables are data structures that consists of *weak pairs*. In a weak pair, the first element (the key) is kept by a weak reference while the second element (the value) is kept by and ordinary reference. In the Java programming language, a weak table is implemented by the class `WeakHashMap`, using weak references. In Lua, a weak table is a primitive mechanism (which can be used to implement regular weak references when needed). However, in Lua, we can construct not just weak tables containing weak pairs but also weak tables with strong keys and weak values or both weak keys and values.

In Lua, each table has a metatable that controls its behavior. Each field in the metatable controls a specific aspect of the table. The field `__mode` is a string that controls the weakness of the table. If this string contains the character 'k', the keys are weak; if it contains the character 'v', the values are weak; and if that string contains both 'k' and 'v', both keys and values are weak. Code 1 shows how to create a weak table consisting of weak pairs in Lua.

Code 1 Creating a weak table in Lua.

```
a = {}          -- a regular table
b = {}          -- its metatable
setmetatable(a, b)
b.__mode = "k"  -- makes 'a' weak
```

Most of weak references' typical uses involve weak tables. From weak references' (and weak tables') most relevant applications, we highlight the following:

– **Collection of cyclic data structures** - Pure reference-counting garbage collectors are unable to reclaim cyclic data structures. According to Brownbridge [Brownbridge 1985], this deficiency was the initial motivation for the development of weak references. It can be easily overcome by replacing ordinary references with weak ones in a way that every cycle has at least one weak reference. With the advent of tracing garbage collectors this use is becoming obsolete.

– **Cache** - Applications that make frequent use of large data structures may improve their performance significantly by keeping these structures resident in memory. However, this can lead to fast memory exhaustion. Weak tables provide a simple solution to implement automatically managed caches that preserve data only as long as memory is not scarce.

  According to the literature [Ierusalimschy 2006, Jones et al. 1999], a common use of caches are memoized functions. The computational effort necessary to process a complex function can be significantly reduced by saving its results. When the function is reinvoked with the same argument, it just returns the saved value. In a long running application the storage used by memoized functions can grow to prohibitive levels. The use of weak tables in this case can transparently preserve the most recently (and probably the most frequently) accessed values without compromising memory availability.

– **Weak sets** - Weak sets can be understood as a set of objects whose pertinence to this set does not lead to a reference to the object. Weak sets can be used as an implementation solution whenever objects must be processed as a group, but without interfering with their life cycles. A common example is the *Observer* design pattern, which defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [Gamma et al. 1995]. This notification is done by the observed object that needs to know its observers. In a loosely coupled application, these references must not prevent the observers collection. The use of a weak table to keep the set of observers does not protect them from being collected.

– **Finalization** - Weak references extended with a notification mechanism can be employed to inform a client program that an object was collected, eventually executing routines associated with such events. This is called container-based finalization and contrasts with class instance finalizers, which implement object-oriented finalization [Hayes 1997, Jones et al. 1999].

– **Property tables** - Weak tables can be used to add arbitrary properties to an object, i.e., they can represent property tables. The benefit of using weak tables to represent a property table relies on the fact that, in most cases, adding a property to an object should not modify the time of its collection. For example, consider the table in Figure 1. In this table, each key has a weak reference to an object, and each respective value has an ordinary reference to the extra properties of this object. If references from keys to objects were ordinary, the simple fact of inserting a new key/value pair in the table would prevent the collection of the object referenced by the key. When the references from keys to objects are weak, the objects can be collected as soon as they are no longer used by the client program.

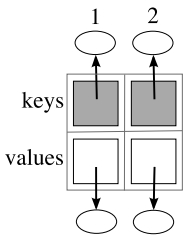

Figure 1: A property table

Ideally, a key/value pair in a weak table must be kept only as long as its key is reachable, directly or indirectly, from some place outside the table. (A key is directly reachable if it is referred by some outside object. It is indirectly reachable if it is referred by some value whose key is directly or indirectly reachable.) However, most programming languages' implementations do not present this behavior. A problem occurs when the value part of some entry has a direct or indirect reference to a key, forming a cycle inside its table or a cycle between elements of different weak tables. The table depicted in Figure 2 shows this problem. Due to the self-reference of element 1 (value points to its key) and to the cycle created by elements 2 and 3, the objects referenced by these elements will never be collected. Even when there are no cycles, the collection of objects can take more time than expected. Consider the elements 4 and 5, where the value of element 4 has a reference to the key of element 5. Generally, the garbage collector is able to remove element 5 only in a cycle subsequent to the one where element 4 was removed. A weak table with a chaining of n elements will take at least n cycles to be entirely collected. One may think that changing the values from strong to weak will help, but this is not correct. Consider a property table and an object that exists only as a value in this table. If the reference from the value to the object is weak, this means that the object can be collected. However, the corresponding key may be active and the search for the value (object) using the key is still possible.

Cycles can also appear, for example, when a weak table is used as a cache that handles memoized functions. Suppose we want to create constant functions: given a value, create a function that always returns this value. An implementation for this function, in Lua, could be:

```
function K (x)
    return function () return x end
end
```

If we want to memoize this function (so that it's not necessary to recreate this function for an already treated value), we need a table mapping `x` (the key)

to `K(x)` (the value). Notice that the value is a function that contains a reference to `x`. This way, neither the key nor the value will ever be collected.

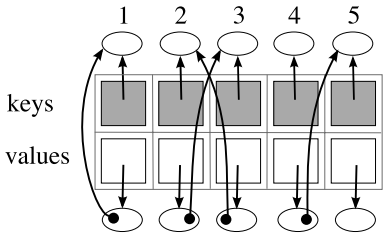

Figure 2: Problems with weak pairs.

Programming languages like Java and Lua face the problem of cycles in weak tables. In Java's `WeakHashMap` API there is an implementation note stating that "care should be taken to ensure that value objects do not strongly refer to their own keys, either directly or indirectly, since that will prevent the keys from being discarded" [SUN 2006].

3 Ephemerons

An interesting solution to the problem of cycles in weak tables, first presented by Hayes [Hayes 1997], is the use of *ephemerons* instead of weak pairs. Ephemerons are a refinement of weak pairs where neither the key nor the value can be classified as weak or strong. The connectivity of the key determines the connectivity of the value, but the connectivity of the value does not affect the connectivity of the key. According to Hayes, when the garbage collection offers support to ephemerons, it occurs in three phases instead of two (mark and sweep). We consider next the first and second phases, as they are the most relevant for our work. Details of the third phase can be found in the work of Hayes [Hayes 1997].

In the first phase, the graph of objects is traced, following the references among them. Whenever an ephemeron is found, instead of tracing the fields of the ephemeron, the collector inserts the ephemeron in a queue to be processed later. The ephemerons in this queue may or may not contain reachable entries.

In the second phase, the collector scans the queue of ephemerons. When the key of an ephemeron has already been reached the corresponding value is traced - if the key is reachable then some part of the program may ask for the value. Any ephemeron whose key has not been reached may or may not contain a reachable value; these ephemerons are kept in the queue for future inspection. Now we have two group of ephemerons, those who have reachable keys and those who

have not. The first group is traced in the same way any other non-ephemeron object group would be. Because the key has been reached, and traced, only the value need to be traced. However, the values can contain references to keys of some ephemerons still in the queue, what will make these keys reachable. When such a thing happens, the queue needs to be inspected again. Besides that, extra ephemerons can be found. In this case, they are inserted into the queue.

This procedure continues until the queue has only ephemerons whose keys has not yet been reached. When the queue converges to such a set, the collector can recycle the memory occupied by these ephemerons. A pseudo-code for the second phase's algorithm is presented in Code 2.

Code 2 Second phase of the ephemeron's collection algorithm

```
1: while TRUE do
2:   for all e ∈ ephemeron-queue do
3:     if key(e) is marked then
4:       put(reachable-value, e)
5:     else
6:       put(unreachable-value, e)
7:     end if
8:   end for
9:   ephemeron-queue ⇐ unreachable-value
10:  if reachable-value is not empty then
11:    for all e ∈ reachable-value do
12:      trace(value(e))
13:    end for
14:    make-empty(reachable-value)
15:  else
16:    return
17:  end if
18: end while
```

The Haskell programing language defines a semantics for weak references based on key/value pairs that is similar to ephemerons. Our work shows that it is also possible to easily incorporate ephemerons to the present implementation of the Lua programing language. This is done by modifying a small part of the garbage collection algorithm. The next section describes how we implemented the ephemerons mechanism and measured its efficiency and effectiveness.

4 Eliminating Cycles

As we saw in Section 2, most weak references' uses involve weak tables. However, the problem of cycles imposes an obstacle in the use of weak tables, as it can cause loss of memory or a delay in the memory's recycling. Although not well known in the programming languages community, the ephemerons mechanism is a very reasonable solution for this problem. Therefore, we decided to adapt Lua's garbage collection algorithm to offer support to this mechanism, measuring the impact of doing so.

Basically, an ephemeron consists of a key/value pair; these pairs are not necessarily stored in a table. However, because we need ephemerons whenever facing cycles in weak tables, it is reasonable to think about *ephemerons tables* instead of individual key/value pairs. Ephemerons tables is a good way of implementing ephemerons in Lua because a table is Lua's main data structure and because Lua offers support to weak references via weak tables. In our approach, a weak table with weak keys and strong values is an ephemerons table. Instead of including a new type of table, we simply changed the garbage collection algorithm. That way, in the new implementation, to create an ephemerons table, you just need to create a weak table with weak keys and strong value in the usual way. Section shows an efficiency measurement between the old implementation of this kind of weak table and the new implementation, where it means an ephemerons table.

Now let us see how it works. Code 3 shows a simple cycle being created inside table `et`, an ephemerons table. In this cycle, the first entry's value has a reference to the second entry's key. This cycle would never be collected if a weak table with weak keys was used. After creating the cycle, there is a explicitly call for the garbage collection. Because we are using an ephemerons table, the cycle will be collected and, at the end of the execution, the value of the variable `count` will be zero.

4.1 Implementation

Before discussing the implementation, we will take some time to describe the Lua garbage collector.

The Lua garbage collector implements the *tricolor marking* incremental algorithm [Dijkstra et al. 1978]. This algorithm interleaves the tracing phase, when (non)garbage is detected, with the programs execution [1]. In the tricolor marking algorithm, the collector can be described as a process that traces the objects assigning colors to them. In the beginning, all objects are colored white. As the collector traces the graph of references, it assigns black to each object traced.

[1] For more information on garbage collection algorithms and techniques refer to the work of Wilson [Wilson 1992].

Code 3 Example of how an ephemerons table works.

```
et = {}
mt = {}
setmetatable(et, mt)
-- sets the table to be an ephemerons table,
-- in the new implementation.
mt.__mode = "k"

a = {}
b = {}
et[a] = b
et[b] = a
a = nil
b = nil
collectgarbage()
count = 0
for k,v in pairs(et) do
count = count + 1
end
print(count) -- 0
```

By the end of the collection, the objects accessible to the program must be colored black, and all white objects are removed. However, because the program's and the garbage collector's executions are interleaved, we also need to consider the intermediate phases. Furthermore, the program cannot be allowed to change the graph of references in a way that the collector will fail to find all reachable objects. To prevent this, the tricolor marking algorithm uses a third color, grey, to represent reached objects whose descendants may not have been traced yet. This way, whenever an object is found during tracing, it is colored grey. When all descendants are traced the object is colored black. Intuitively, the tracing proceeds in a wavefront of grey objects which separates white (unreached) objects from black ones, with an invariant that a black object never refers a white one. However, the program can still need to create a pointer from a black to a white object, which violates the invariant. To prevent this, the collector uses a *write barrier*: whenever the program attempts to write a reference to a white object into a black object, either the origin changes (from black) to gray or the destination changes (from white) to gray.

The Lua garbage collection is divided in four phases. In the first phase, the collector traces the graph of references coloring the objects. This phase is interleaved with the program's execution. The second phase is called the atomic

phase, when several operations are executed in a single step - some garbage collector's operations cannot be interleaved with the program's execution (e.g., the clearing of weak tables). In the third phase, also incremental, the collector frees the memory occupied by white objects. Finally, in the fourth phase, the finalizers are called interleaved with the program's execution. To simplify the understanding of the garbage collector's behavior, we will consider two phases in our discussion: an atomic phase, consisting of the former second phase, and a non-atomic phase, consisting of the first, third and fourth phases.

In the ephemerons' collection algorithm, the garbage collector first traces the graph of objects. When it finds an ephemeron, instead of immediately tracing the ephemeron's fields, the garbage collector inserts this ephemeron in a queue to be processed later and carries on the tracing phase. In our case, we needed a data structure to represent a queue of ephemerons table. The original implementation of the Lua garbage collector has one single data structure to store all three kinds of weak tables, the list `weak`. In our implementation of the garbage collector, this list was divided into three separate lists, one for each kind of weak table. The list that stores weak tables with only weak keys is called `ephemeron`, as all tables of this kind signify an ephemerons table. This way, during the tracing phase, whenever the garbage collector finds a ephemerons table it inserts this table into the list `ephemeron`.

We added new actions to the garbage collector's phases in order to implement the ephemerons mechanism. First, as described earlier, whenever an ephemerons table is found it is enlisted in `ephemeron`. The entries from the ephemerons tables are not traced, neither the keys nor the values. (Remember that each key/value pair represents one ephemeron in the original mechanism.) This is all that was added to the first phase.

Afterwards, the collector enters the atomic phase, when several operations are executed in a single step. We implemented two new functions for this phase: `traverseephemeron`, that traces the list `ephemeron`, and `convergeephemerons`, that calls the first function in order to converge the list `ephemeron`. A pseudo-code for these functions can be found in Code 4. The function `traverseephemeron` traces all the entries from an ephemerons table. If some entry's key is marked, meaning that it was reached, the corresponding value is marked. The function `traverseephemeron` returns a boolean value, defined by the variable b in Code 4. This boolean is true if any value from the ephemerons table has been marked, and false otherwise.

The function `convergeephemerons` calls `traverseephemeron`. If this function returns true, i.e., if some value has been marked, the former calls a function from the collector's original implementation, `propagateall`. This function was not modified. Its responsibility is to trace the program's graph of references and, according to the tricolor marking algorithm, to expand the barrier of grey

objects. Note that, in the previous execution of `traverseephemeron`, values in an ephemerons table were marked. Therefore, `propagateall` will mark objects referenced directly or indirectly by these values. Because these new marked objects may refer to keys in ephemerons tables, the function `convergeephemerons` calls again `traverseephemeron`. When no value is marked, `traverseephemeron` returns false. If after tracing all ephemerons tables no value was marked, then `convergeephemerons` terminates its execution. The functions `convergeephemerons` and `traverseephemeron` adapt the second phase of the original ephemerons mechanism to the Lua garbage collector.

Code 4 Pseudo-code for `convergeephemerons` and `traversephemeron`

```
function convergeephemerons(ephemeron)
 1: while TRUE do
 2:    changed ⇐ FALSE
 3:    for all e ∈ ephemeron do
 4:       if traverseephemeron(e) then
 5:          changed ⇐ TRUE
 6:          propagateall(...)
 7:       end if
 8:    end for
 9:    if not changed then
10:       break
11:    end if
12: end while

function traverseephemeron(e)
1: b ⇐ FALSE
2: for all pair ∈ hash(e) do
3:    if key(pair) is marked then
4:       mark the value
5:       b ⇐ TRUE
6:    end if
7: end for
8: return  b
```

After the execution of `convergeephemerons`, and still in the atomic phase, the collector calls the function `cleartable` — as `propagateall`, this function also belongs to the collector's original implementation. The function `cleartable` removes unreachable entries from weak tables, including ephemerons tables which are weak tables with weak keys and strong values. Finally, the collector proceeds

to its last phase, when it deallocates memory and executes the finalizers. This phase was not modified.

5 Efficiency Analysis

In this section we analyze the collection of ephemerons tables and the collection of weak tables with only weak keys, as it was implemented in the original garbage collector.

Consider a program A that creates K_e ephemerons tables, and a program B that creates K_f weak tables. Program A uses the adapted garbage collector, where weak tables with weak keys and strong values signify ephemerons tables, and program B uses the original garbage collector. Suppose every ephemerons table in A has e_h entries in its hash part and e_a entries in its array part [2]. Also suppose that every weak table in B has f_h entries in its hash part and f_a entries in its array part. And consider $e_n = e_a + e_h$ and $f_n = f_a + f_h$.

When dealing with ephemerons tables or weak tables, some garbage collection functions are crucial to the collection's performance. In the adapted garbage collector, these function are:

- `traversetable`: traces a table marking its keys and values;
- `traverseephemeron`: trace an ephemerons tables marking the values whose respective keys were previously reached;
- `cleartable`: removes the entries from ephemerons tables and weak tables that were not reached during the tracing phase.

From these functions, only `traversetable` is executed in the non-atomic phase. This function is executed once for each ordinary, weak tables with only weak values, or ephemerons table. When the table is a ephemerons table, `traversetable` pass the control to `traverseephemeron`. This last function has two loops, one to iterate over the array part and another to iterate over the hash part of the table. Because all numeric keys are strong, all the values in the array part are marked. In the hash part, whenever a key is marked, the value is marked, but the value is not traced. The cost of `traversetable` for program A is $\mathcal{O}(e_n)$. The difference between the adapted and the original garbage collector is that in the later there is no `traverseephemeron` function and `traversetable` traces all tables. This way, the cost of `traversetable` for program B is $\mathcal{O}(f_n)$.

After tracing weak tables and ephemerons tables, the atomic phase of the new garbage collector executes `convergeephemerons`, which continuously call `traverseephemeron` until no value is marked. Now we need to consider the best

[2] Each table in Lua has two parts: an array part indexed by natural numbers, and a hash part indexed by other objects.

and worst case of `convergeephemerons`. In the best case, there are no values pointing directly or indirectly to keys in an ephemerons table. In that case, there are up to two iterations over the list `ephemeron`: one to mark the values whose keys have been reached and a second iteration in case some value was marked in the first one. The cost of `convergeephemerons` in the best case (for program A) is $\mathcal{O}(K_e \times e_n)$.

The worst case happens when there is a key-value chain. The first example of this kind of chaining is depicted in Figure 3. Suppose the client program has a strong reference to the first value from the first ephemerons table. Note that a value from each ephemerons table points to the key of the next table, forming a chain. In this case, the loop in `convergeephemerons` is executed $K_e + 1$ times: one time to mark each value and one last time that modifies nothing. In this example of worst case the cost of `convergeephemerons` is $\mathcal{O}(K_e^2 + (K_e \times e_n))$.

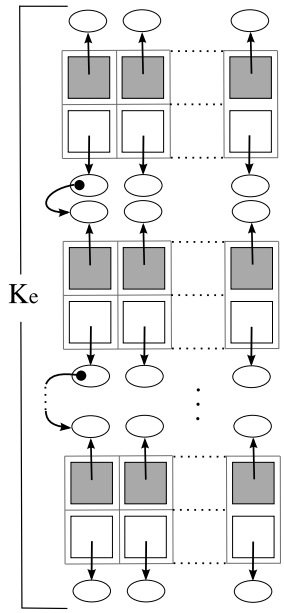

Figure 3: Chaining between tables.

Another example of worst case is depicted in Figure 4, where the keys and values from the same table are chained. The loop in `convergeephemerons` will be executed $(2 \times e_h) + 1$ times: one time to mark each value and a last time that modifies nothing. This way, the cost of `convergeephemerons` is $\mathcal{O}(e_n^2 + (K_e \times e_n))$.

The function `cleartable` has the same behavior in both implementations. This function's cost is $\mathcal{O}(K_e \times (e_n))$ for program A and $\mathcal{O}(K_f \times (f_n))$ for program

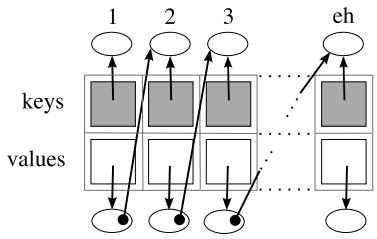

Figure 4: Chaining between keys and values.

B. As a result, the collection's cost for ephemerons tables in program A is $\mathcal{O}(K_e \times (e_n))$ in the best case and $\mathcal{O}(K_f \times (f_n))$ in program B. Consider programs A and B identical, with the exception that A uses the adapted garbage collector. In this case, the collection's cost for each program is the same.

However, in both example of worst case, the collection's cost for program A is quadratic. To be more precise, for the case in Figure 3, the cost is $\mathcal{O}(K_e \times (e_n))$, and for the case in Figure 4 the cost is $\mathcal{O}(e_h \times (e_n))$. The occurrence of chaining between tables or keys and values are rare, but when it happens can affect the efficiency significantly. Because program B uses the original garbage collector, it remains linear. However, notice that cycles in weak tables are not collected in B, possibly causing memory loss (that can be much worse than efficiency loss).

5.1 Efficiency Measurements

We executed two tests in order to measure the efficiency of the extended garbage collector. The first test compares the collection of ephemerons tables without cycles (or chaining) and ephemerons tables with chaining, as shown in Figure 3. We started with different amounts of ephemerons tables without cycles, ranging from 100 to 1000 tables. Each table had 500 entries. After executing the test for ephemerons tables without cycles, we created the same amounts of ephemerons tables with chaining and tested the collection of them. Therefore, we could compare the best and worst case's execution time. The result is depicted in Figure 5. As expected, the curve representing the ephemerons tables with chaining is similar to a quadratic function's curve. The garbage collector's efficiency is highly affected in the worst case. However, because we are using ephemerons tables and not weak tables, all cycles are collected.

The purpose of the second test is to compare the efficiency of the adapted garbage collector with the efficiency of the original one. This time, the weak tables with only weak keys, considered as ephemerons tables in the adapted garbage collector, have no cyles. This test was executed for the same amounts of tables used in the first test, at first just with weak tables (original implementation) and then just with ephemerons tables (adapted implementation). The

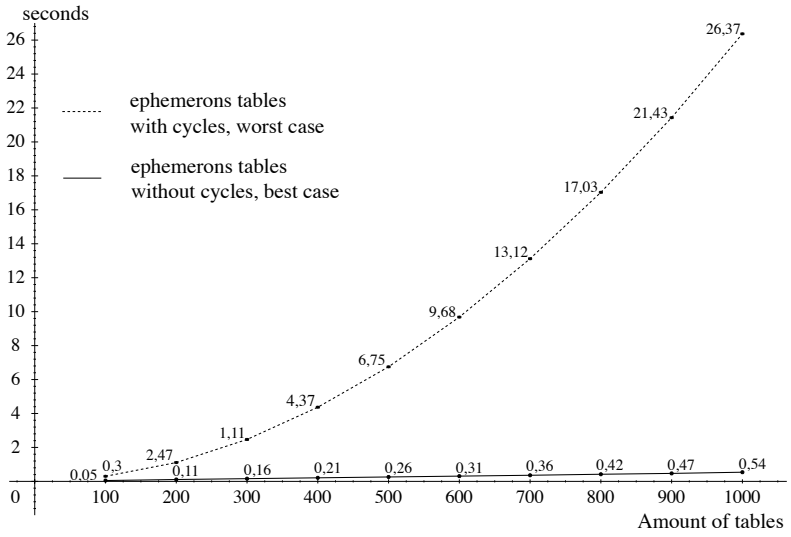

Figure 5: Collection of ephemerons tables: worst case x best case

result is depicted in Figure 6. Notice that there is almost no difference between the time for collecting ephemerons table without cycles and the time for collecting weak tables. We believe that the better result for collecting ephemerons tables is due to some noise in the test's execution, and not to a greater efficiency in the ephemerons tables' collection. We can conclude that, in the lack of cycles, our implementation of the ephemerons mechanism is as efficient as the weak tables' implementation.

6 Conclusion

Weak tables constitute a good way of implementing weak references. In fact, most weak references' uses, such as property tables, weak sets, and caches, involve weak tables. However, the problem of cycles in weak tables yet persists in most programming languages. This can cause loss of memory or a delay in memory recycling. Although not well known among programmers, the ephemerons mechanism is a very reasonable solution for this problem.

Because ephemerons are necessary only in the context of weak tables, we think it is more reasonable to think about ephemerons tables instead of independent key/value pairs. We could easily implement an adaptation of the original ephemerons mechanism, using ephemerons tables, in the Lua garbage collector. In the absence of cycles, our implementation is as efficient as the weak tables' implementation. And when cycles do exist, an ephemerons table is able to collect them. To conclude, we think that this mechanism can be easily implemented in any programing language, as it was done with Lua.

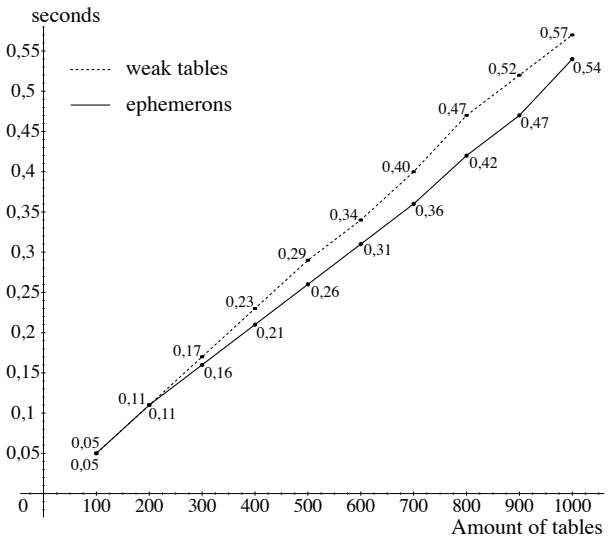

Figure 6: Collection of weak tables x collection of ephemerons tables

7 Acknowledgments

We would like to thank the reviewers for their valuable suggestions.

References

[Atkins and Nackman 1988] Atkins, M. C., Nackman, L. R.: "The Active Deallocation of Objects in Object-Oriented System"; Software: Practice and Experience, 18, 11 (1988), 1073-1089.

[Boehm 2003] Boehm, H.: "Destructors, Finalizers, and Synchronization"; Proc. of the 30th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (2003), 262-272.

[Dybvig et al. 1993] Dybvig, R. K., Bruggeman, C., Eby, D.: "Guardians in a Generation-based Garbage Collector"; Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (1994), 207–216.

[Hayes 1992] Hayes, B.: "Finalization in the Collector Interface"; Proc. of the '92 International Workshop on Memory Management, London (1992), 277–298.

[Leal 2005] Legal, M. A.: "Finalizadores e Referências Fracas: Interagindo com o Coletor de Lixo"; PhD thesis, Informatics Department, Pontifical Catholic University of Rio de Janeiro (2005).

[Bloch 2001] Bloch, J.: "Effective Java Programming Language Guide"; Prentice Hall, first edition (June 2001).

[PARC 1985] Xerox Palo Alto Research Center (PARC): "InterLISP Reference Manual"; Palo Alto, CA (October 1985).

[Rees et al. 1984] Rees, J. A., Adms, N. I., Meehan, J. R.: "The T Manual"; Yale University, Computer Science Department, fourth edition (January 1984).

[SUN 2006] Sun Microsystems: "Java^{TM} Plataform Standard Edition 6.0: API Specification"; (2006).

[Ierusalimschy 2006] Ierusalimschy, R.: "Programming in Lua"; Lua.org, second edition (2006).

[Hayes 1997] Hayes, B.: "Ephemerons: a New Finalization Mechanism"; In Proc. of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, New York, NY (1997), 176–183.

[Jones et al. 1999] Jones, S. P., Marlow, S., Elliott, C.: "Stretching the Storage Manager: Weak Pointers and Stable Names in haskell"; Lect. Notes Comp. Sci. 1868, Springer (1999), 37-58.

[Glasgow 2007] The Glasgow Haskell Compiler user's guide, version 6.2; `http://www.haskell.org/ghc`, last viewed in April 2nd, 2007.

[Brownbridge 1985] Brownbridge, D. R.: "Cyclic Reference Counting for Combinator Machines"; Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture, New York, NY (1985), 273–288.

[Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J.:"Design Patterns: Elements of Reusable Objetct-Oriented Software"; Addison Wesley (1995).

[Dijkstra et al. 1978] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., Steffens, E. F. M.: "On-the-fly Garbage Collection: An Exercise in Cooperation"; Communications of the ACM, 21, 11 (November 1978, 966-975.

[Wilson 1992] Wilson, P. R.: "Uniprocessor Garbage Collection Techniques"; Proc. of the 1992 International Workshop on Memory Management, Saint-Malo, France (1992), 1-42.