TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

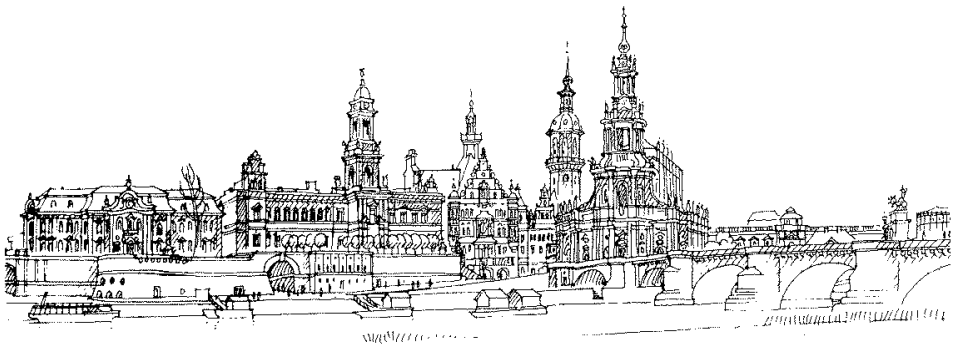
Technische Berichte
Technical Reports
ISSN 1430-211X

TUD-FI09-11 November 2009

Thomas B. Preußer, Peter Reichel
Rainer G. Spallek

Institut für Technische Informatik

An Embedded Garbage Collection Module with Support for Multiple Mutators and Weak References



Technische Universität Dresden
Fakultät Informatik
D–01062 Dresden
Germany
URL: http://www.inf.tu–dresden.de/

A Fully-Concurrent Non-Blocking Embedded Garbage Collection Module with Support for Multiple Mutators and Weak References

Thomas B. Preußer Peter Reichel Rainer G. Spallek

Institut für Technische Informatik
Technische Universität Dresden
01062 Dresden, Germany

{*thomas.preusser, rainer.spallek*}*@tu-dresden.de*
peter@peterreichel.info

Abstract *This report details the design of a garbage collection (GC) module, which introduces modern GC features to the domain of embedded implementations. The described design supports weak references and feeds reference queues. Its architecture allows multiple concurrent application cores operating as mutators on the shared memory managed by the GC module. The garbage collection is exact and fully concurrent so as to enable the uninterrupted computational progress of the mutators. It combines a distributed root marking with a centralized heap scan of the managed memory. It features a novel mark-and-copy GC strategy on a segmented memory, which thereby overcomes both the tremendous space overhead of two-space copying and the compaction race of mark-and-compact approaches. The proposed GC architecture has been practically implemented and proven using the embedded bytecode processor SHAP as a sample testbed. The synthesis results for settings up to three SHAP mutator cores are given and online functional measurements are presented. Basic performance dependencies on the system configuration are evaluated.*

1 Introduction

The automatic reclamation of memory space no longer in use is an essential feature of modern software platforms. Widely known as garbage collection (GC), it completely drains a prominent source of programming errors and dramatically increases programmer productivity. In spite of its inherent costs in processing time and memory bandwidth, it has even been adopted in the embedded platform domain. Also there, the increasing system complexity calls for higher programmer productivity and increased product confidence.

In addition to general time-to-market and design cost requirements, a few additional design constraints are of prominent importance especially in the embedded platform domain. Embedded systems should be frugal and just offer the needed performance with the smallest feasible silicon and the least possible power consumption. However, the specified performance is also often strongly required to be delivered so that services are guaranteed to meet their deadlines. These aspects motivate the designated and optimized implementation of essential functionalities in spite of the additional design effort. The garbage collection is one example for such an optimization target, which has already been approached by several research groups.

While garbage collection automates the management of one essential system resource, the heap memory; many other resources (files, locks, queues, sockets etc.) remain subject to the careful manual management by the programmer. Implementors of the representing objects of such resources usually seek to automate the proper closing of the resource with the help of the automated memory management. Due to the indeterminism of many GC techniques, this does not compensate for a missing programmer-triggered release of the resource immediately after its use but it is an appropriate fallback ensuring an eventual clean shutdown of the resource. The classical means to achieve this coupling was the finalizer, which, however, suffers several issues in its application:

- It may be overridden and, thus, even be deactivated by a careless programmer not calling the inherited implementation of an extended class.
- The order of finalizer invocation and their executing threads are undetermined.
- Exceptions thrown during finalizer execution are ignored and do not generate any feedback whatsoever.

These issues are solved by the use of weak references in conjunction with reference queues. This concept combines a clean collection of no longer referenced objects with a notification of the mutator, which is achieved by enqueuing the associated reference objects. These objects act as proxies representing the weak references to their targets without keeping them alive. The number of weak references to an object and, thus, the number of notifyable observers is only bounded by the system resources rather than the concept itself. The observation of its lifecycle does not require a cooperative implementation of the target, which eases the use of this design pattern and eliminated potential pitfalls. The processing of proxies enqueued to the notification queues is under the sole control of the queue creator.

Besides the advantages over classical finalizers, weak references even without associated reference queues enable new flavors of reachability. These can be used to mark certain associations as *nice to have but not critical*. This allows the caching of larger reconstructible data while still allowing the GC to reclaim the occupied memory whenever needed.

Both finalization and weak references enjoy thorough support in the Java2 Standard Edition (J2SE). Weak references even come in three flavors: soft, weak and phantom, which support caching as well as pre-finalization and post-finalization notification. In the Java2 Micro Edition (J2ME), only the Connected Device Configuration (CDC) requires the same set of features. The Connected Limited Device Configuration (CLDC) for more constrained devices only knows about the single weak flavor of references and adopted it only with version 1.1. It avoids finalization totally. Thus, it is not at all surprising that hardware GC or hardware-assisted GC implementations do normally not support any notion of weak references or even classical finalization.

This report proposes a designated concurrent hardware GC architecture that supports weak and soft references as well as reference enqueuing as known from the J2SE. Without finalization support, the phantom references collapse semantically with the weak ones[1]. This makes them obsolete in a CLDC setting.

The proposed GC implements a copying approach within a segmented memory so as to reduce the 100 percent space overhead of classical copying GC significantly. It further employs designated tap units on the internal mutator components to collect the root reference set without explicit mutator assistance.

In the remainder of this report, Sec. 2 gives an overview on the previous work undertaken on hardware-assisted GC implementations. Sec. 3 describes the proposed GC architecture. Its

[1] In contrast to the other flavors of weak references, the Java API particularity requires the *unretrievable* target of a phantom reference to be cleared programmatically in order to become *unreachable* in spite of an alive phantom reference to it. This particularity appears to stem from the avoidance of a software patent (`http://forums.sun.com/thread.jspa?threadID=5230019`). It, in fact, prevents a complete semantic blend of weak and phantom references, which is, however, of no conceptional importance.

practical implementation is evaluated by Sec. 4 based on its integration into SHAP [19, 27]. Sec. 5, finally, concludes this report.

2 Related Work

Systems with minimal hardware support for their GC implementations were built already in the late 1970s and early 1980s as native LISP machines [8,12,13], which featured type-tagged memory words to identify pointers and hardware-implemented read and write barriers.

The later Garbage Collected Memory Module by Nilsen et. al [14–17, 21] implements an object view upon the memory space. Upon request from the mutator, which needs to provide a list of the root objects, it employs a copying approach for an independent garbage collection. Although well documented, this system has never been prototyped [11].

Srisa-an et. al [24, 25] describe a mark-and-sweep accelerator backed by allocation bit vectors. Additional 3-bit reference counting within similar bit vectors is suggested for the fast reclamation of low-fanin objects. The scalability of this approach is achieved by the caching of partial vectors as directed by an undetailed software component. Also, this architecture was merely simulated as C++ model and has not been included in any synthesized design.

The Komodo [18] architecture and its successor jamuth [26] support a very fine-grained, low-cost thread parallelism on instruction level. This architecture enables the non-intrusive execution of background system threads, one of which can be designated to the garbage collection. Different implementations of mark-and-sweep algorithms based on DIJKSTRA's tri-color marking are described.

Meyer [9] described a RISC architecture, which features a separate register sets and storage areas for pointers and other data. This enables a clear distinction among these data types as required for an exact garbage collection. A microprogrammed co-processor, finally, implements the actual GC with a copying approach [10]. The required read barrier is later backed by designated hardware so as to reduce its latency significantly [11].

Gruian and Salcic [5] describe a hardware GC for the Java Optimized Processor (JOP) [22,23]. The implemented mark-and-compact algorithm requires the mutator to provide the initial set of root references. As objects may be moved during the compaction phase, read and write accesses to them must be secured by appropriate locks. The mutator may otherwise proceed with its computation concurrently to the garbage collection. Reference and data fields inside heap objects are distinguished by appropriate layout information associated with each object's type. Root references on the stack are determined conservatively only excluding data words that do not resemble valid reference handles.

In summary, several embedded GC implementations built on designated architectural resources and may thus be called hardware-assisted. The controlling algorithms are typically implemented in software albeit regularly on a level as low as microcode. The implementations are frugal in the sense that they provide an automatic memory management but without any more extras. In particular, they avoid the support of any extended object lifecycle management such as finalization, weak references or even reference enqueuing. These systems are obsolete even for the implementation of the current CLDC 1.1. These drawbacks are targeted by our GC architecture.

3 Garbage Collector Design

3.1 Fundamental Design

The desired GC architecture should provide several functional features. First of all, it is to abstract the memory to a managed object heap with high-level object creation and field access operations as well as automatic garbage collection. It is further to support weak references as defined for the CLDC 1.1. Finally, the capability to attach multiple mutators is desired.

The GC task obviously grows more complex with the support of weak references. In order to keep its implementation maintainable and extensible, the FSM implementation of SHAP's former memory manager [20] was abolished. Instead, several options for software-programmed

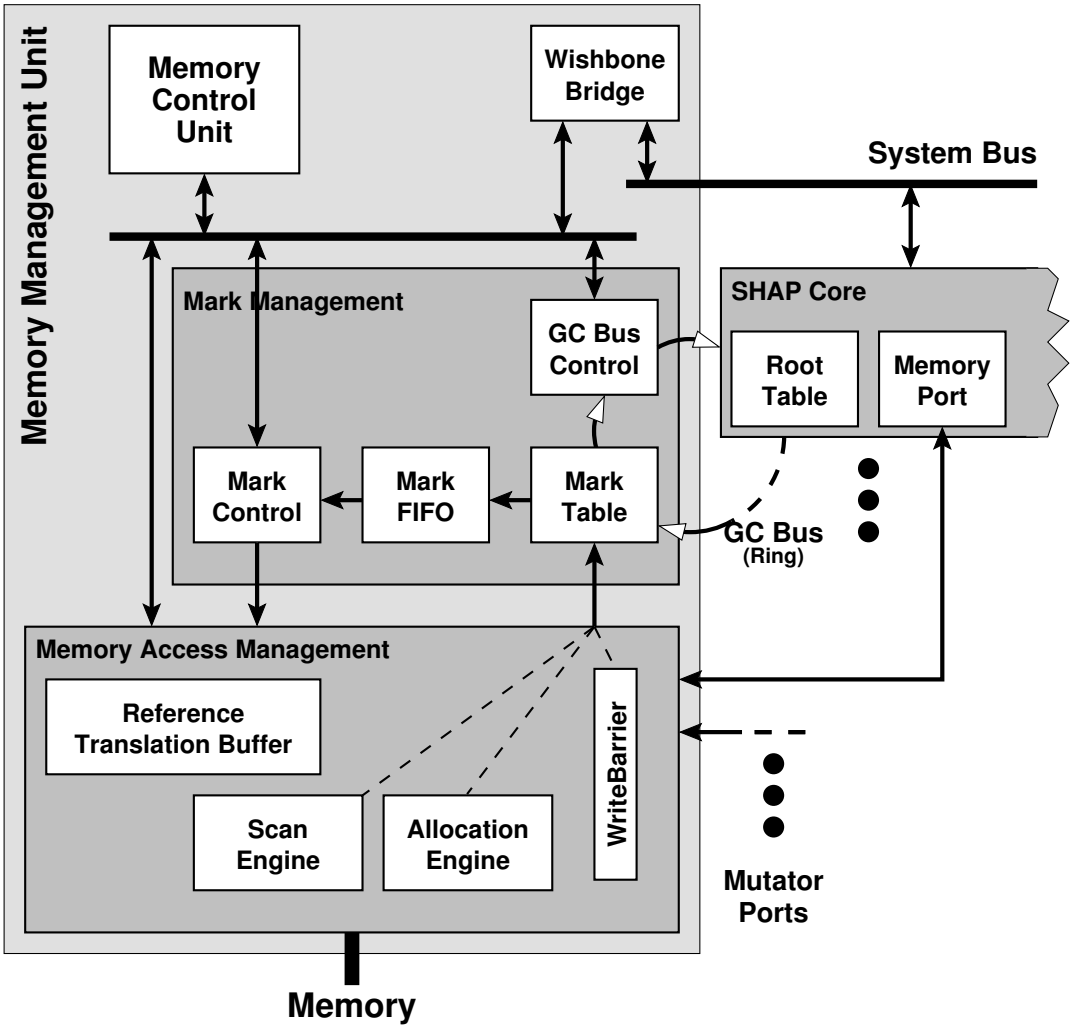


Figure 1: Memory Management Unit: Structural Overview

solutions were explored. The obvious solution to simply duplicate SHAP itself was soon dismissed as its high-level Java programming greatly relies on the memory object abstraction that first needs to be established by this component. The continuous access to low-level memory would further require a unnatural if not abusive use of the language. Last but not least, the microcode implementation used by SHAP would establish a significant overhead when compared to the still rather compact memory management task. A reduced variant duplicating only the core microcode engine, finally, disqualifies due to the low achieved gain in abstraction level. Thus, the search was narrowed to compact RISC cores with a functional C toolchain, specifically, to the OpenFIRE [1] and the ZPU [6].

Both of these cores are freely available but quite contrary in philosophy. While the OpenFIRE is a full-fledged RISC engine, the ZPU takes a very frugal approach. It turned out that the ZPU with a few replacements of emulated by implemented instructions was well-sufficient for the task of memory management. It further provides the strong advantages of a concise design structure, lower resource demand and a high achieved clock frequency. In fact, an OpenFIRE solution would require a second slower clock domain or the reduction of the overall system clock by 40%. Consequently, it was the ZPU microarchitecture, which we chose to be at the heart of our memory management.

As illustrated in Fig. 1, the ZPU assumes the central control of the memory management as memory control unit (MCU). It is

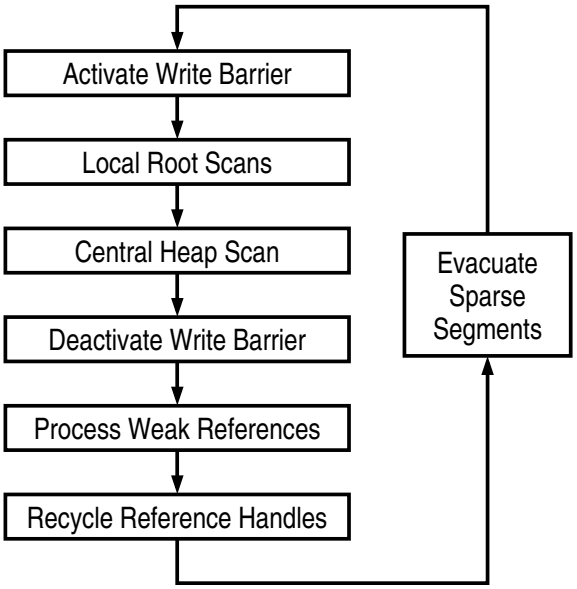


Figure 2: GC Cycle Overview

surrounded by several special-purpose hardware components implementing time-critical subtasks. Their functions will be detailed later. The connection to the system Wishbone bus enables the (slow) administrative communication with the mutator cores. This not only serves the communication of statistics data but is also used for delivering weak reference proxies to the runtime system for their possible enqueuing into reference queues so that our design supports this feature in addition to the CLDC requirements.

The memory access of the mutator cores are prioritized over GC-related accesses. The garbage collector, thus, operates on cycle stealing, a very fine-grained utilization of otherwise idle memory bandwidth enabled by the mutator-independent concurrency of the memory management unit.

The overall GC cycle is summarized in Fig. 2. It is initiated and supervised by the MCU. Some tasks are, however, backed by dedicated hardware components containing small specialized state machines.

3.2 GC Strategy

The GC strategy should enable a smooth turnover of memory in object allocation and recycling. In order to guarantee an instant allocation of objects, we chose a bump-pointer allocation scheme simply forwarding the allocation pointer by the number of the requested words. Obviously, such an approach requires the compaction of the used storage to re-generate the continuous allocation region. The two established alternatives are the single-phase copying garbage collection and the two-phase mark-and-compact approach.

The traditional copying garbage collector [7] uses two memory spaces to encode the liveness of an object by its physical storage location. With alternating roles, the current *to-space* contains newly-allocated and other reached objects transferred from *from-space*. Allocation in *to-space* is performed continuously. While not requiring a designated mark phase, this approach regularly moves the whole living object graph through memory without allowing any old stable part of it to settle somewhere. The requirement of one empty memory half establishes an expensive overhead.

The classical mark-and-compact approach builds the graph of living objects within a designated mark phase. This is succeeded by the compaction phase, which condenses the living objects, say, at the bottom of the memory. The allocation of new objects can continue right behind this condensed storage area. In a concurrent GC implementation, this approach may suffer from a race between allocation and compaction where each allocation adds to the compaction work and, thus, delays the re-generation of the allocation area. This approach was implemented for JOP's hardware-assisted GC [5].

While both of these approaches incur undesirable drawbacks in their pure adoptions, these can be mitigated by the partitioning of the available memory into disjunct, typically equally-sized segments. From these, an initially empty segment is designated to the allocation of new objects, which is performed in a simple bump-pointer manner. When the space within this segment does no longer suffice the allocation of a requested object, a hardware FIFO provides the descriptor of an immediate replacement segment so that no time-consuming intervention of the MCU is required. It only becomes responsible for used segments. It regularly initiates a heap scan to detect the objects no longer in use and to update the segment utilization statis-

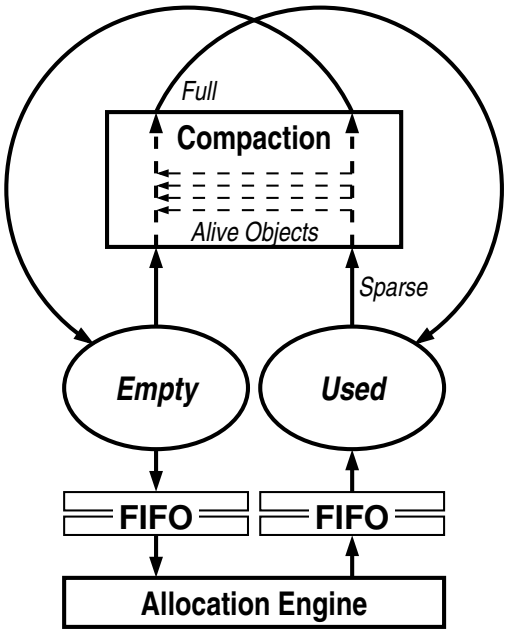


Figure 3: Segment Life Cycle

tics accordingly. It also selects segment with a low remaining utilization to be cleaned out. All its remaining living content is copied compactly into an evacuation segment before the cleared segment becomes available as empty storage. The MCU will substitute the evacuation segment by a new empty one as soon as it fails to receive an object surviving the collection of its home segment.

The life cycle of the memory segments is illustrated in Fig. 3. While the general descriptor-based administration of the segments is performed by the MCU, there are specialized hardware state machines for time-critical tasks. The allocation engine reserving memory space for new objects inside the memory access management is even decoupled by FIFOs so that the independent instant replacement of a filled allocation segment is possible. Also the object movement as part of the compaction process is implemented as service provided by the memory access management. This low-level integration of this process enables transparent mutator access even to objects being copied. In contrast to JOP's hardware GC, the mutator need not protect object accesses by locks. Even if the accessed object is currently being moved, the access will be guided correctly to either the old or the new copy according to the instant position of the copying pointer within the object.

The proposed algorithm establishes a novel mark-and-copy approach operating on a segmented memory. It enables continuous allocation and concurrent garbage collection. A race between allocation and collection has been avoided as both are operating in distinct segments. The copying effort is reduced to surviving objects co-residing with garbage in the same segment. Segments with only short-lived operational objects are freed as a whole without any copying work. Segments with accumulated old long-lived objects will remain untouched. In addition to the spontaneous formation of object generations, the collector can accelerate this trend by the use of generational evacuation segments.

The critical parameter of the proposed approach is the segment size. Firstly, it restricts the size of the largest allocatable object. Secondly, small segments increase the the management overhead in terms of segment exchanges and state information. On the other hand, large segments force a coarse-grain memory management with a potentially significant space overhead approaching the behavior of a copying collector. Hence, a set of well over 4 segments should be, at least, available.

Having decided for a moving GC, measures must be taken to ensure the stable identification of each object throughout its life cycle even in the possibility of its displacement. This is achieved through fully-transparent handles, which are the only identifications of objects ever known to a mutator. The memory manager internally maps a handle to a state record comprising the current storage location of the referenced object, the sizes of its reference and data areas as well as some GC information.

3.3 Exact Garbage Collection

Targeting SHAP, a secure embedded bytecode processor, it is imperative to implement an *exact* garbage collector. Such a collector must be able to safely distinguish references to heap objects from primitive data words as to obtain a *precise* picture of the boundaries of the graph of living objects. While *conservative* collection

approaches are equally *safe* in the sense that they will never discard objects that are, in fact, still in use, they may fail to collect applicable garbage. Only relying on heuristics to exclude definite non-references (such as bit pattern signatures), unfortunate primitive values mirroring valid references are capable to increase the maintained object graph to zombie structures beyond the living core. This bears the risk of a decreasing system performance and even of undue memory exhaustion. An exact GC, on the other hand, ensures that the memory overhead of an application as compared to its true requirement is only induced by the collection latency, i.e. the collection cycle time. Garbage not discovered due to unfortunate data words does not exist.

The exact distinction between primitive data and references is achieved through additional administrative metadata. This may either be provided through high-level structural information on the object layouts or by a low-level tagging of individual memory words. While type-tagged memory was, indeed, used by the early LISP machines named above, the modern established word-based memory interfaces with corresponding standard software data types do no longer provide the space even for a single tagging bit. The overhead of a parallel tag memory in a separate memory region or even as physical module should be avoidable on a platform where all memory objects are laid out after the blueprints of a handful of classes.

Meyer's RISC architecture [9] offers a suitable implementation of a rigorous spatial separation of primitive data and references into two storage area within an object. It does, however, not cope well with inheritance. As a specialized subclass may arbitrarily add primitive data or reference fields to the state of instances, both of the storage areas must be able to grow. Nonetheless, any object must still be viewable as an instance of any superclass. Consequently, at least one of the two storage areas must be accessed in two sequential steps: (1) query the runtime size of the other storage area to be skipped, and (2) access the field at the given index behind this area. An approach without this deficiency had, however, already been proposed

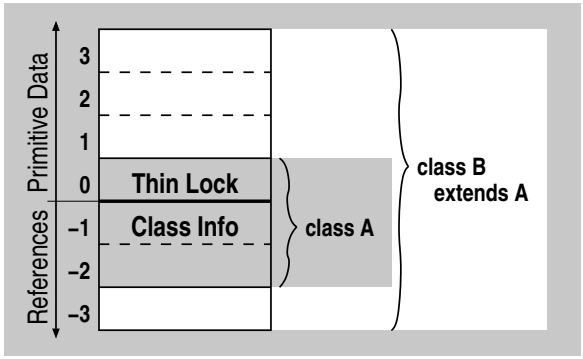


Figure 4: Bidirectional Object Layout

for the SableVM [4]: using a bidirectional class layout, the index ranges of the fields in both storage areas can grow independently. Thus, all field accesses can be performed by a single indexed memory access taking the central object pointer as the base address. Although described for a software implementation, this approach is well suited for its adoption in a hardware architecture.

The distinction between primitive data and references must, however, be extended beyond the actual garbage-collected heap to values held by the mutator cores. These values contain the set of roots to the living object graph residing in memory. Since local registers and the execution stack are typically a rather loose collection of individual values, their tagging is a valuable option. As no established external interfaces are effected, this option is also feasible. Tagging is, indeed, not as rigorous as the separated register sets as suggested by Meyer's RISC architecture. On the other hand, it is more flexible in the use of its internal storage resources avoiding workload effects such as unbalanced register pressure due to architectural usage constraints. For our adaptation for SHAP, we, thus, chose to employ reference tagging inside the mutator cores. There, it is applied to registered values and the integrated execution stack.

For our garbage collector, we adopted the bidirectional object layout for heap objects as illustrated by Fig. 4. The object layout is specified when the runtime and application classes are linked for SHAP. The linker separates primitive data and reference fields assigning them indices growing independently in magnitude in

declaration order. Static field values are held accordingly within the Class objects representing their defining classes. While all arrays are dimensioned on demand, the elements of reference arrays grow into the reference area while the elements of primitive array extend into the primitive data area. These measures ensure the following heap properties:

1. Each value in the reference area of a living object is either `null` (encoded as zero(0)) or a legal reference to a living object.

2. Memory outside the reference areas of the living objects will not be scanned by the garbage collector. Hence, the contained values cannot keep any objects alive.

While the mutator should generally not transform among primitive data and references, such operations may be quite useful, for instance, for the derivation of an identity hashcode. Only one thing must be strictly forbidden: a mutator may never *invent* a reference handle unless it can prove that this reference is already existing and alive. The allocation of objects and the assignment of reference handles as well as the recycling of handles fallen out of use is the sole responsibility of the memory management.

3.4 Object Graph Marking

A GC cycle must calculate the subgraph of reachable objects rendering all other objects unreachable and, thus, no longer usable by a mutator so that there occupied memory can be reclaimed. An object is reachable by a mutator if it is referenced (a) directly by the mutator or (b) transitively by a field of a reachable object. The set of objects directly reached through mutator references is commonly referred to as the root set.

The reachable object subgraph can be computed simply by following its definition: After querying the mutators for their held roots, a graph search from the known reachable objects is performed to explore the whole reachable subgraph. The technical challenge of this procedure is its execution concurrently to the mutators so as to allow their continuous computational progress and the concurrent modification of the root sets and of the object graph. The ongoing alteration of the object graph somewhat blurs the computation of its reachable subgraph. Nonetheless, the following properties must be met:

Correctness All objects that are reachable when the computation completes belong to the computed subgraph.

Boundedness No object that was never reachable throughout the computation belongs to the computed subgraph.

While the correctness ensures that a garbage collector is safe to use, its boundedness defines its very purpose and ensures that all objects that are unreachable when a GC cycle starts will be reclaimed.

Our architecture builds upon a decentralized collection of the root set. Each mutator core is extended by its own root scan unit, which collects the references contained in the local registers and stack into a mark table. As shown in Fig. 1, all mutator cores are arranged in a ring on a GC bus, which is mastered by the MMU. This unit issues commands onto the ring and receives their acknowledgements as they return on the other end. This ring also serves the gradual merger of the contents of the individual root tables into the global root set. While the MMU transmits an empty table, each core `OR`s the received table with its own table contents before forwarding it along the ring. The final results is entered into the central mark table inside the MMU.

The chosen ring structure has several advantages over a direct connection of all root scan units to the central mark table. First of all, it guarantees a simple routing of rather short signal paths only connecting ring neighbors. In conjunction with the local root tables, it further allows the root scan to be performed concurrently on all mutator cores without congestion in the access to the central mark table.

The local root scans operate concurrently to their associated mutator core. Once finished, they are deactivated for the remainder of the GC cycle so that further modifications of the local root sets are not logged. Not *unmarking* lost (overwritten) roots does not violate correctness

nor boundedness as these objects were reachable when the GC cycle began. Not marking new roots is also legal as they are either (a) references loaded from memory and, thus, reachable from the original root set or (b) newly-allocated and, then, implicitly marked by the allocation engine.

After the completion of all root scans, the actual computation of the reachable subgraph is performed by the heap scan, which is an optimized implementation of the well-established tri-color marking [2]. While the mark table entries distinguish reached from unreached objects, the colors *red* or *green* are assigned to references to divide the reachable objects into those already scanned and those still requiring processing. The meaning of the colors alternates and is determined relatively to the color of the active GC cycle. Initially, all references are unscanned and have the color opposite to the current cycle. Upon being scanned, a reference assumes the cycle color. References to newly-allocated objects are immediately assigned the active color as they do not contain valid references and need not be scanned. The heap scan is completed when all reachable and, thus, marked references have assumed the cycle color. After disposing of the remaining unmarked and, thus, unreachable references, the meaning of the colors is exchanged and the initial condition that all references are of the opposite color is re-established for the next GC cycle.

While the completion of the scan could be detected by another walk through the mark table that does not produce any marked reference without cycle color, we implemented a simple but effective optimization. Each time a previously unmarked reference is marked in the mark table, it is also copied into a mark FIFO. This FIFO is read to determine objects that still have to be scanned during the heap scan. Only when it runs empty, the mark table is re-walked to search for additional work but only if the walk just completed did produce a FIFO overflow. As all work has been safely finished when the FIFO sufficed, a final unproductive walk of the table is no longer needed. This implementation is also a significant improvement for settings where the scan of an object always only marks new table entries that have just been passed. Without the FIFO, this could degenerate to require a number of walks through the full table that approaches the table size.

The concurrent modification of the object graph by the mutator must be accounted for so as to avoid the violation of correctness while, of course, only marking objects that were reachable at some time in the course of the scan. In particular, it is necessary to ensure that no mutator ever gets hold of a reference that remains unmarked. This situation might occur when a reference field of a still unscanned object is read and overwritten by a different value before it is scanned. While the mutator then holds a reference into an object subgraph, the reference to be followed by the heap scan has been destroyed. To keep the effected subgraph, nonetheless, alive, a write barrier is used to intercept reference writes and to enter the overwritten reference into the mark table. Although this might, indeed, keep truly unreachable subgraphs alive, boundedness is not violated as the subgraph was reachable at some time during the ongoing scan.

In the context of multiple mutators, the write barrier is implemented using an atomic swap operation on the backing memory as to outrule a race condition among possibly concurrent read-write sequences on the same storage location. Although only required for the heap scan, we chose to activate the write barrier even prior to the initiation of the local root scans. This choice relaxes the phase transition from root to heap scan and avoids its system-wide synchronization through an atomic rendezvous.

3.5 Weak Reference Support

In contrast to their regular *strong* counterparts, weak references do not keep their referents alive. An object may become eligible for garbage collection in spite of the existence of paths via weak references to it. If the collector decides to discard the referent of a weak reference, the reference must be deprived of its capability to retrieve the referent, which is usually achieved by clearing it to `null`. Reference queues may establish an additional notification scheme allowing cleared references to be enqueued for processing by an application thread.

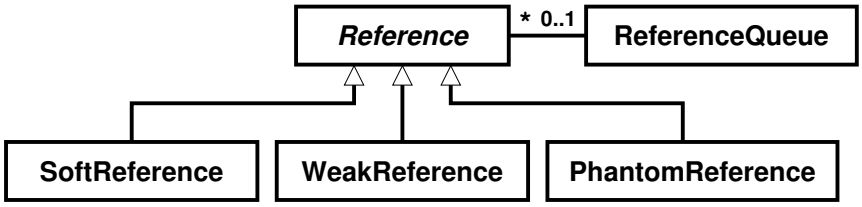


Figure 5: Hierarchy of Reference Proxy Classes

The Java 2 Standard Edition knows of several different strengths of weak references:

1. Soft references have a rather strong grip on their referents. Even if an object is only reachable going through a soft reference, the garbage collector is urged not to sacrifice the object and clear the soft references as long as the system has sufficient memory.

2. Weak references do not at all keep their referents alive. As soon as the GC determines an object to be solely weakly reachable, it will be discarded and the weak references will be cleared.

3. Phantom references are a particularity that allows the ultimate detection of the death of an object. Phantom references are only enqueued after all other references have been cleared and the finalizer of the object has been run.

As shown in Fig. 5, all of them derive from their common superclass `Reference` and any instance of these references is applicable for the use with a reference queue, which is specified at construction time. Phantom references are not cleared automatically[2] and require a reference queue as polling their referent using `get()` returns `null` by definition so that the queue establishes the only useful communication channel.

From this spectrum, the CLDC 1.1 of the Java 2 Micro Edition only requires the implementation of the weak references. Note that, without the support for finalization, the role of phantom references is assumed by the weak references and their absence, thus, not a loss. Soft

[2] Apparently due to a software patent, c.f. `http://forums.sun.com/thread.jspa?threadID=5230019`

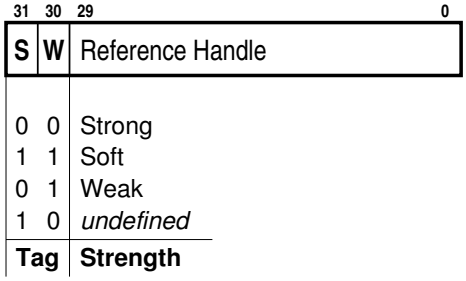


Figure 6: Tagging of Reference Strengths

references and reference queues are, however, additional useful features, which we decided to provide in the GC architecture for SHAP albeit they are not required by the current CLDC.

Weak references are typically implemented as proxy objects containing a special reference member initialized to the referent from a regular strong reference provided to the constructor. The garbage collector must be enabled to recognize these special references in order to treat them appropriately in the heap scan and to clear them when it is about to collect their referents. This distinction is typically achieved through the type of the proxy object so that the garbage collector must recognize all subclasses of these proxies and be aware of the special references contained at a certain field offset.

The necessity to provide the GC with information about the class hierarchy and the object layout of reference objects is not desirable. We avoided it by tagging the weak references themselves. As the exact garbage collection has been achieved through the bidirectional object layout, signature bits assisting the conservative garbage collector are no longer needed. Instead, the memory manager uses two bits to distinguish different strength of references as shown in Fig. 6. While these bits are generally cleared, they are set by the constructors of reference

objects. Although not currently used, this approach enables a much more flexible use of weak references even outside the hierarchy of the reference objects.

References of different strengths define different degrees of reachability of the objects contained in the managed heap. If these were to be calculated during the heap scan, the scan time would potentially multiply as parts of the objects graph are re-scanned in order to update the reachability information after finding a stronger reference to a subgraph root. Fortunately, this effort is not necessary. Weak references are simply *not* followed during the heap scan. Merely, their containing proxies are collected. After the heap scan, the list of proxies is scanned for references to unreached referents. Any one found will be cleared, and the effected proxy will be communicated to the mutators for its possible enqueuing into a reference queue. Thus, the weakly-reachable parts of the object graph are simply not scanned and may then be regularly discarded.

Soft references assume a special role as they should only be followed as long as there is plenty of memory available in the system. Therefore, their treatment is decided in the beginning of a garbage collection cycle according to the current memory utilization. If memory is abundantly available, they are treated like strong references, otherwise they are treated like weak references and are not followed during the scan.

Special care is to be taken upon the retrieval of a strong reference from a weak proxy. Assume that an object only remains reachable through a weak reference. As long as the garbage collector does not discover this condition, the reference is not cleared but totally valid. The invocation of the `get()` method on the `WeakReference` proxy object will return with a normal strong reference to the referent effectively resurrecting the object from the brink of death. A race condition may now arise when such a resurrection interferes with an ongoing heap scan, which would normally ignore the weak reference. Hence, it must be ensured that the reference to be returned by `get()` is either entered into the mark table prior to the completion of the heap scan or invalidated by returning `null`. Not knowing whether an ongoing heap scan will render the referent strongly-reachable or not, a consistent resurrection with a mark table entry is attempted first. Only if the scan has finished in the meanwhile, the actually determined reachability is evaluated. If necessary, `null` will be returned in conformance to the inevitable clearing of the original reference field.

While the memory manager collects cleared weak reference objects, they must be processed further by the mutators. For this purpose, the runtime system forks a designated service thread that maintains the communication with the memory manager via its Wishbone connection to receive the list of cleared proxies of a collection cycle. It is the responsibility of this runtime service to enqueue these proxies appropriately if they have associated reference queues.

4 Evaluation in SHAP

The described design was implemented for SHAP and integrated into the runtime system through adapted microcode implementations accessing the port to the memory access manager as well as through regular Wishbone I/O for less time-critical operations. The latter also provides fundamental statistical data that we used for this evaluation. The runtime library was extended to enable application access to the new features and to provide implementations for the reference proxy and the reference queue classes. It was also turned into the first client of the weak reference support by the implementation of resource pools, which, for instance, enable the proper interning of strings.

The reference design was implemented on a Xilinx Spartan-3 XC3S1000, which is capable of holding up to three SHAP cores next to the memory management unit. The utilization of this reference platform is summarized in Tab. 1. As indicated, the demand on active resources grows linearly with the number of integrated SHAP cores. While most of the basic core-independent flip-flops and LUTs may be attributed to the central memory management, global Wishbone-attached IO accounts for about a quarter of it. The three Block RAMs outside the cores are

Table 1: Device Utilization of Reference Platform

Cores	**FFs**	**LUTs**	**BRAMs**
1	3206	7813	10
2	4465	11396	17
3	5720	14963	24
n	$\approx 1257n + 1950$	$\approx 3575n + 4240$	$7n + 3$

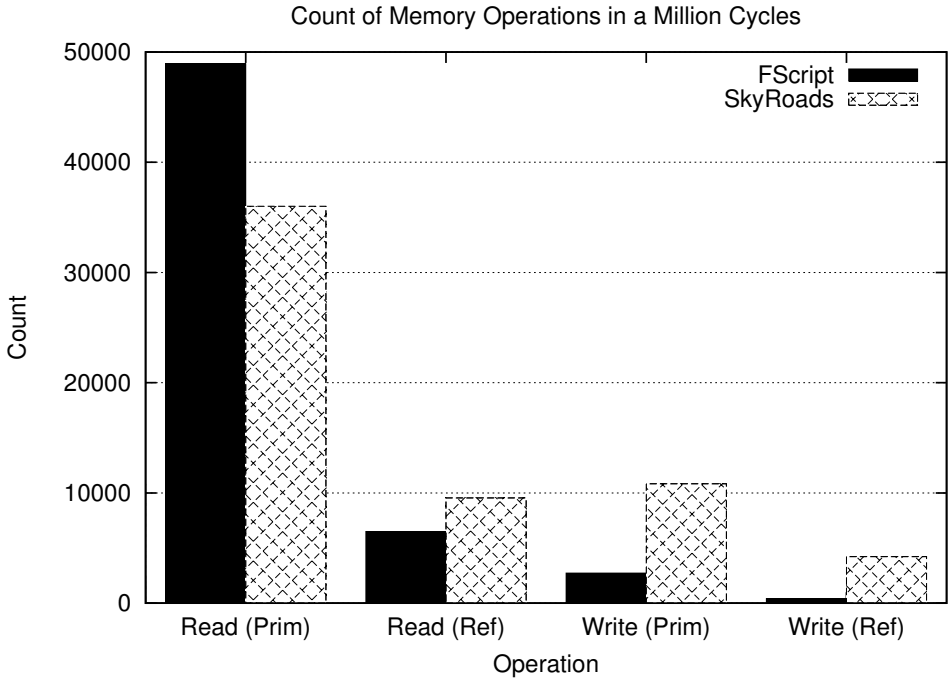


Figure 7: Frequency of Read and Write Operations

used as MCU storage also containing the GC program and the global mark table.

While we chose a write barrier to capture changes to the object graph in the course of the heap scan, a read barrier would be equally valid. Marking every reference read, it would ensure that all subgraphs a mutator obtains a reference into will be kept alive even if the read reference becomes overwritten later. The decision in favor of the write barrier is motivated by the significantly lower frequency of writing as compared to reading memory accesses as shown in Fig. 7 for two sample applications running on SHAP and also reported in earlier quantitative analyses [3]. Thus, the choice of a write barrier reduces barrier activations and, hence, the competition for mark table access.

The processing of living weak reference proxies constitutes a processing overhead as they are enlisted during the heap scan to be revisited thereafter in order to verify that their referents have been reached or to clear and enqueue them. This overhead grows linearly with the number of living proxy objects. This is shown in Fig. 8 for several scans of a heap with an identical object population, which merely differs in the number of weak reference proxies pointing to living objects rather than being cleared. While the heap scan merely slows down marginally, the overhead becomes clearly visible in the succeeding weak reference processing. The lower impact on the heap scan is due to its hardware-assisted implementation inside the memory access management.

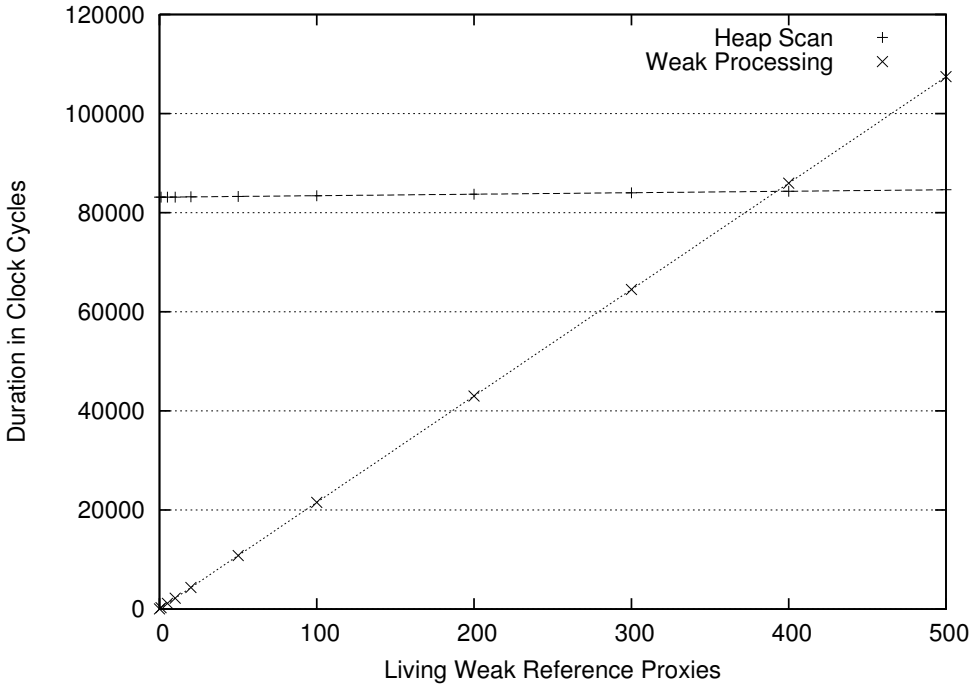


Figure 8: Overhead of Weak Reference Processing

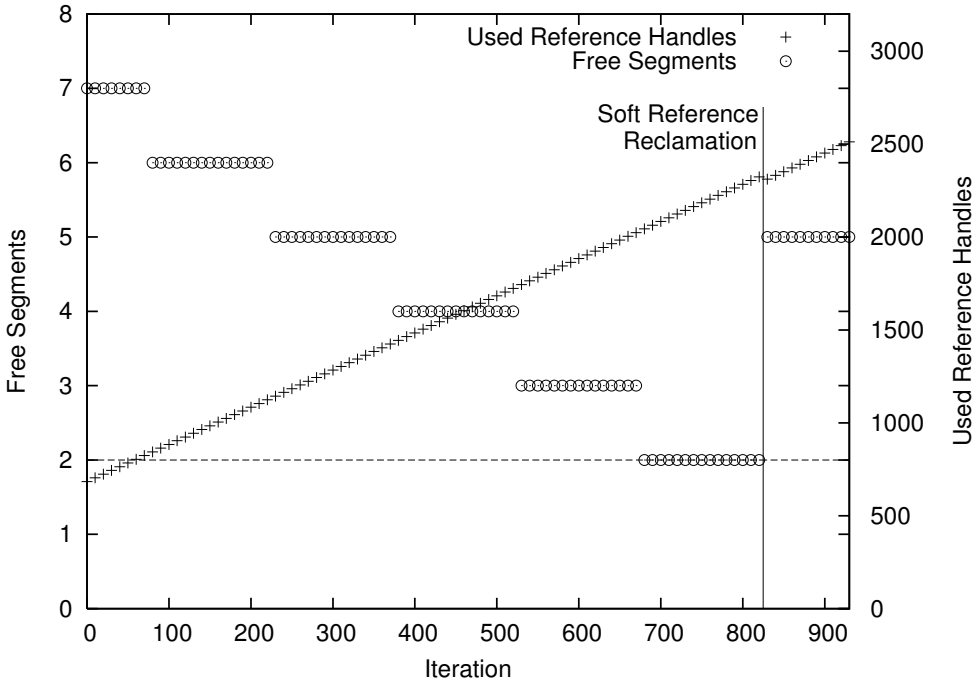


Figure 9: Soft References and Growing Memory Utilization

The distinguished treatment of soft references according to the current memory utilization is illustrated in Fig. 9. Initially, a set of few but large objects is allocated and made solely soft-reachable. Then, small objects are created continuously and kept reachable so that the memory is filled up. While the large objects are initially kept alive, the available empty segments will eventually fall short of the threshold configured to two segments. When this happens, the

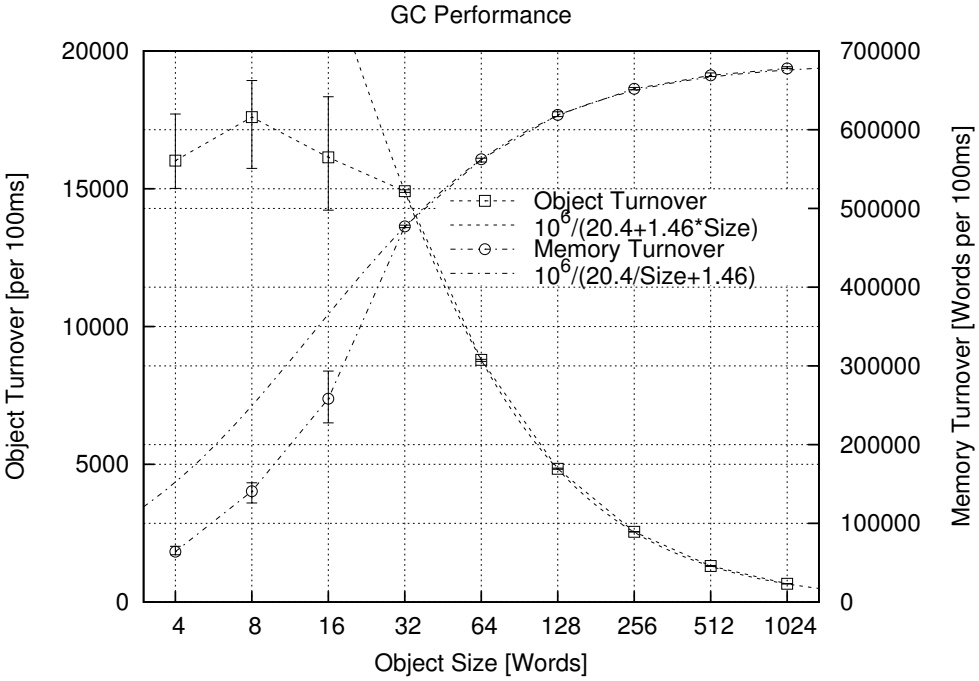


Figure 10: GC Performance

GC no longer follows soft references and will, thus, collect all solely softly-reachable objects. This reclamation of soft references is clearly shown by the discontinuity in Fig. 9. There, the collection of the large softly-reachable objects frees a large chunk of three complete memory segments while merely recycling a couple of reference handles.

Also, the performance of the garbage collector was evaluated within a SHAP system running at 50 MHz on the mentioned Spartan-3 FPGA with an attached 1 MiByte SRAM, of which about 50 KiByte are occupied by API and application code. The benchmark application spawns two threads. The first of these constantly allocates heap objects of a pre-defined size, which are abandoned immediately after their allocation to become eligible for garbage collection. The second thread only wakes up every 100 ms just to log the current object allocation count, which is incremented by the first thread upon each successful object creation.

The measurements were performed for different object sizes. Each measurement spans over the arbitrary number of eight consecutive 100 ms intervals. In order to determine the sustained turnover of objects and memory, which includes both their allocation and their recycling, these eight intervals are preceded by another one to reach the desired saturation. The reference system configuration provides a pool of 2^{13} reference handles and a memory space consisting of 2^{18} 32-bit memory words.

The obtained object and memory turnovers are plotted against the respective object sizes in Fig. 10. Performance jitter between the measuring intervals is only observable for object sizes below 32 words and is depicted by the dispersion bars. Although small, this variation can be directly attributed to an increased interference of the GC module with the mutator core for small objects. The performance-limiting factor in these cases is apparently the pool of available object handles rather than the actual memory consumption. Also, the fairly constant object turnover for object sizes below 32 words suggests that the exhaustion of handles rather than memory forces the GC into collection cycles.

As the object size grows beyond 32 words, the object turnover decreases notably. In this region, the performance is limited by the SHAP core itself. Implementing a Java platform, it has to zero the memory of a freshly allocated object before initiating its construction. Thus,

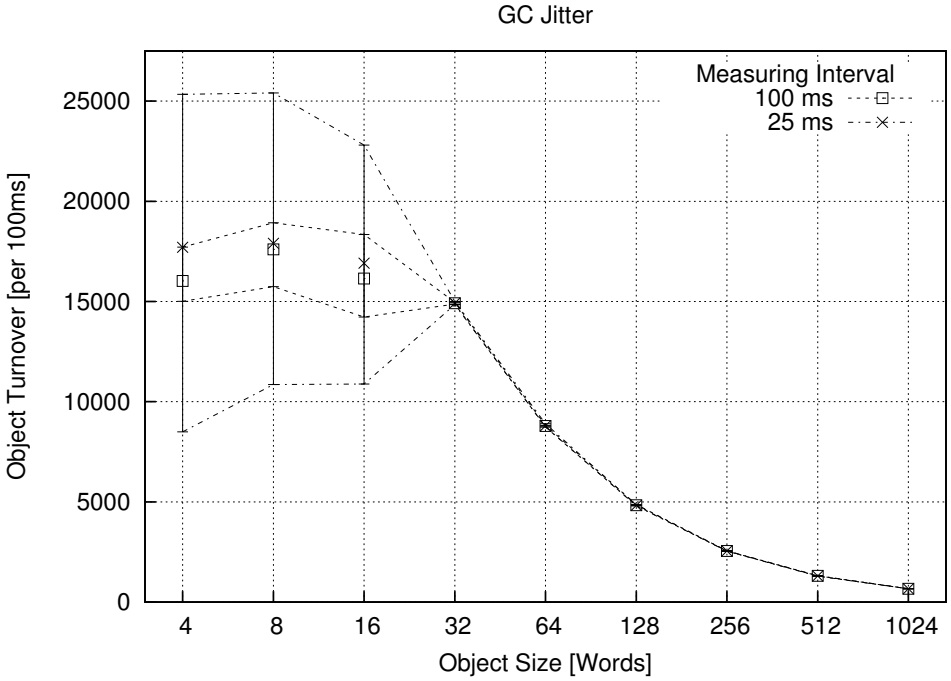


Figure 11: GC Jitter

the memory turnover, i.e. the object turnover multiplied by the object size, is bounded by the speed of zeroing. Merely the constant loop and construction overhead for larger objects allows a steady but slowing increase of the memory turnover and prevents the object turnover from halving with every doubling of the object size. This dependency of the object turnover from the object size is illustrated through its good approximation by the given hyperbola. While the numerator of this approximation is an arbitrary scale to the GC performance, the constant summand 20.4 and the coefficient 1.46 of the denominator represent the constant overhead and cost per object word, respectively. The corresponding approximation of the memory turnover, which is simply scaled by the object size, is also shown.

Fig. 11 provides some deeper insight into the interference of the garbage collection with the mutator progress. By the shortening of the measuring intervals, the averaging effect hiding the jitter in computational progress is reduced. This is clearly apparent in the normalized object turnover graphs for the interval length of 25 ms. But again, such jitter can only be observed for small object sizes below 32 words. It completely disappears for larger object sizes. Special care is, thus, required when tasks with short periods produce much garbage consisting of many small objects. Otherwise, the concurrent garbage collections can successfully operate in the background.

As shown in Fig. 12, the memory turnover suffers when the memory size decreases, here to half the space. The exhaustion of the memory available for allocation is underlined by the increased performance jitter even beyond object sizes of 32 words. This shows that the garbage collector can no longer operate solely in the background but has to draw memory bandwidth from the mutator in order to sustain the required collection speed.

Finally, Fig. 13 shows the effect of doubling the number of available reference handles. Supporting the thesis of an exhausted handle pool for small object sizes in the initial measurement, the object turnover is clearly increased in this region while the figures for larger object sizes remain unaffected. Notably, the object turnover slightly decreases for an object size of 32 words at the borderline of these regions. Apparently, the increased reference pool incurred an additional small but measurable processing cost.

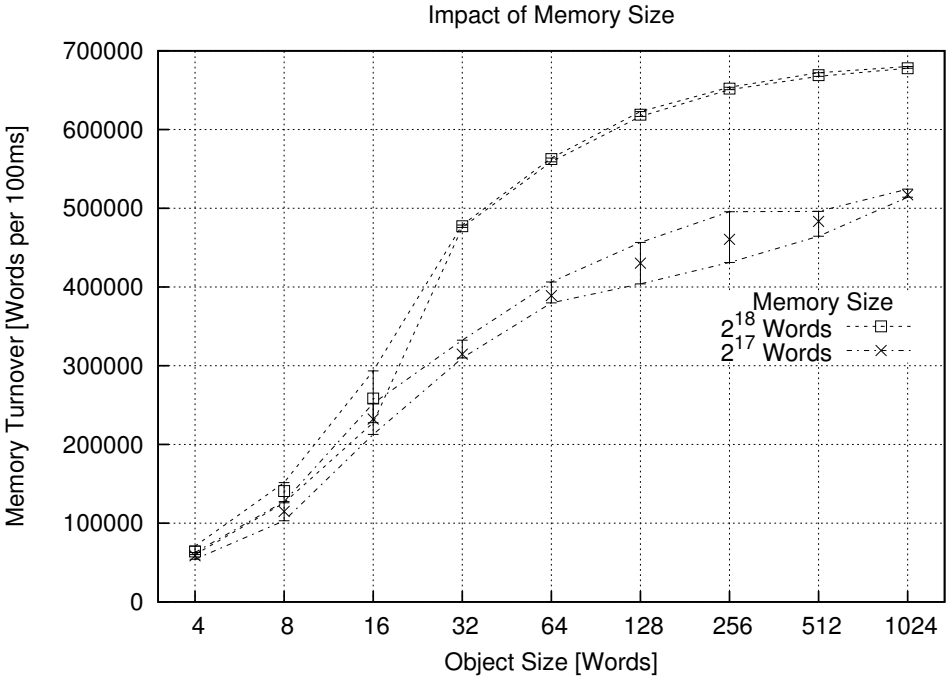


Figure 12: Impact of Memory Size

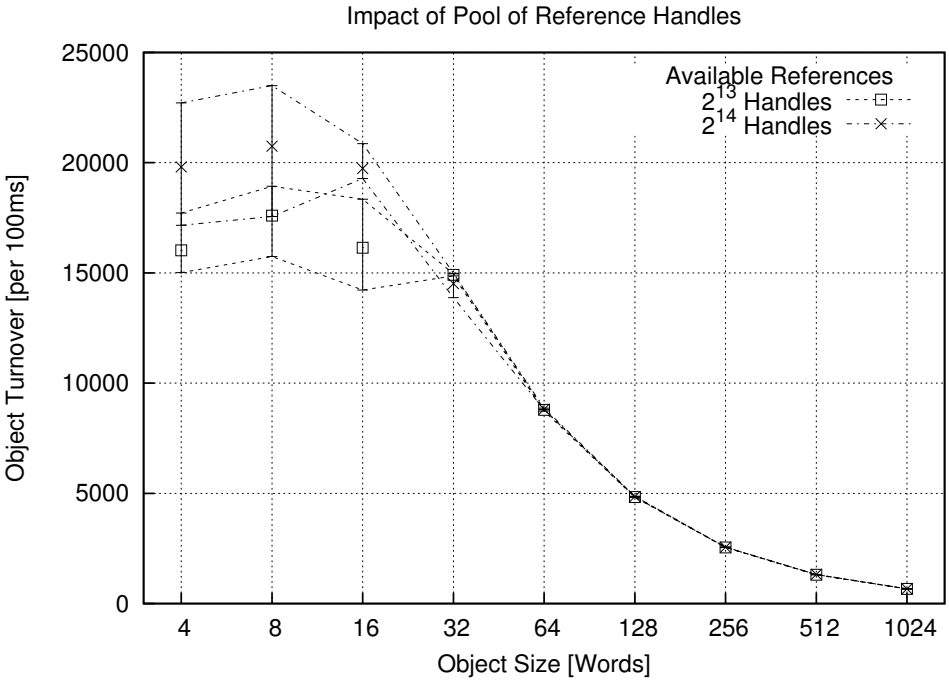


Figure 13: Impact of Pool of Reference Handles

5 Conclusions

This report has demonstrated that the integration of advanced GC features is feasible even for small embedded bytecode processors. The presented solution makes thorough use of hardware acceleration wherever applicable while employing a main C-programmed software control for easy maintenance. The presented solution implements a concurrent non-blocking garbage collector with the support for multiple mutators. Even going beyond the requirements of the CLDC 1.1, it includes the support for soft references and reference enqueuing. The practical implementation of the garbage collector has been proven and evaluated in the SHAP bytecode processor. Basic dependencies of the GC performance of the system configuration have been illustrated and evaluated.

References

[1] A. Anton. OpenFIRE, 2007. `http://www.opencores.org/project,openfire2`.

[2] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[3] M. W. El-Kharashi, F. ElGuibaly, and K. F. Li. A quantitative study for Java microprocessor architectural requirements. Part II: high-level language support. *Microprocessors and Microsystems*, 24(5):237–250, Sept. 2000.

[4] E. M. Gagnon and L. J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Java Virtual Machine Research and Technology Symposium*, pages 27–40, Apr. 2001.

[5] F. Gruian and Z. A. Salcic. Designing a concurrent hardware garbage collector for small embedded systems. In *Asia-Pacific Computer Systems Architecture Conference*, pages 281–294, 2005.

[6] Ø. Harboe. ZPU - the worlds smallest 32 bit CPU with GCC toolchain, 2008. `http://www.opencores.org/project,zpu`.

[7] J. Henry G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[8] J. Holloway, G. L. S. Jr., G. J. Sussman, and A. Bell. The SCHEME-79 chip. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Lab., 1980.

[9] M. Meyer. A novel processor architecture with exact tag-free pointers. *IEEE Micro*, 24(3):46–55, 2004.

[10] M. Meyer. An on-chip garbage collection coprocessor for embedded real-time systems. In *RTCSA '05: 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 517–524, Washington, DC, USA, 2005. IEEE Computer Society.

[11] M. Meyer. A true hardware read barrier. In *ISMM '06: 5th International Symposium on Memory Management*, pages 3–16, New York, NY, USA, 2006. ACM.

[12] D. A. Moon. Garbage collection in a large LISP system. In *LFP'84: 1984 ACM Symposium on LISP and functional programming*, pages 235–246, New York, NY, USA, 1984. ACM.

[13] D. A. Moon. Architecture of the symbolics 3600. *SIGARCH Comput. Archit. News*, 13(3):76–83, 1985.

[14] K. D. Nilsen. Progress in hardware-assisted real-time garbage collection. In *IWMM '95: International Workshop on Memory Management*, pages 355–379, London, UK, 1995. Springer-Verlag.

[15] K. D. Nilsen and W. J. Schmidt. Hardware support for garbage collection of linked objects and arrays in real-time. In *ECOOP/OOPSLA '90 Workshop on Garbage Collection*, Oct. 1990.

[16] K. D. Nilsen and W. J. Schmidt. Hardware-assisted general-purpose garbage collection for hard real-time systems. Technical Report TR92-15, Iowa State University, 1992.

[17] K. D. Nilsen and W. J. Schmidt. Preferred embodiment of a hardware-assisted garbage-collection system. Technical Report TR92-17a, Iowa State University, Nov. 1992.

[18] M. Pfeffer, T. Ungerer, S. Fuhrmann, J. Kreuzinger, and U. Brinkschulte. Real-time garbage collection for a multithreaded Java microcontroller. *Real-Time Syst.*, 26(1):89–106, 2004.

[19] T. B. Preußer, M. Zabel, and P. Reichel. The SHAP microarchitecture and Java virtual machine. Technical Report TUD-FI07-02, Fakultät Informatik, Technische Universität Dresden, Apr. 2007.

[20] P. Reichel. Entwurf und Implementierung verschiedener Garbage-Collector-Strategien für die Java-Plattform SHAP, 2007.

[21] W. J. Schmidt and K. D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *ASPLOS-VI: 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–85, New York, NY, USA, 1994. ACM.

[22] M. Schoeberl. JOP: A Java optimized processor. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java*, volume 2889 of *LNCS*, pages 346–359. Springer, Nov. 2003.

[23] M. Schoeberl. A Java processor architecture for embedded real-time systems. *J. Syst. Archit.*, 54(1-2):265–286, 2008.

[24] W. Srisa-An, C.-T. D. Lo, and J. Chang. Scalable hardware-algorithm for mark-sweep garbage collection. In *26th Euromicro Conference*, volume 1, pages 274–281, 2000.

[25] W. Srisa-an, C.-T. D. Lo, and J. en Morris Chang. Active memory processor: A hardware garbage collector for real-time Java embedded devices. *IEEE Transactions on Mobile Computing*, 2(2):89–101, Apr. 2003.

[26] S. Uhrig and J. Wiese. Jamuth: an IP processor core for embedded Java real-time systems. In G. Bollella, editor, *JTRES'07*, ACM International Conference Proceeding Series, pages 230–237. ACM, 2007.

[27] M. Zabel, T. B. Preußer, P. Reichel, and R. G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 59–62. IEEE, Aug. 2007.