J Supercomput (2012) 59:1101–1119
DOI 10.1007/s11227-010-0493-0

Achieving middleware execution efficiency: hardware-assisted garbage collection operations

**Jie Tang · Shaoshan Liu · Zhimin Gu ·
Xiao-Feng Li · Jean-Luc Gaudiot**

Published online: 9 November 2010
© The Author(s) 2010. This article is published with open access at Springerlink.com

Abstract Although virtualization technologies bring many benefits to cloud computing environments, as the virtual machines provide more features, the middleware layer has become bloated, introducing a high overhead. Our ultimate goal is to provide hardware-assisted solutions to improve the middleware performance in cloud computing environments. As a starting point, in this paper, we design, implement, and evaluate specialized hardware instructions to accelerate GC operations. We select GC because it is a common component in virtual machine designs and it incurs high performance and energy consumption overheads. We performed a profiling study on various GC algorithms to identify the GC performance hotspots, which contribute to more than 50% of the total GC execution time. By moving these hotspot functions into hardware, we achieved an order of magnitude speedup and significant improvement on energy efficiency. In addition, the results of our performance esti-

J. Tang · Z. Gu
Beijing Institute of Technology, Beijing, China

J. Tang
e-mail: tangjie.bit@gmail.com

Z. Gu
e-mail: zmgu@x263.net

S. Liu (✉)
Microsoft, Redmond, WA, USA
e-mail: shaoliu@microsoft.com

X.-F. Li
Intel China Research Center, Beijing, China
e-mail: xiao.feng.li@intel.com

J.-L. Gaudiot
University of California, Irvine, CA, USA
e-mail: gaudiot@uci.edu

mation study indicate that the hardware-assisted GC instructions can reduce the GC execution time by half and lead to a 7% improvement on the overall execution time.

Keywords Hardware acceleration · Garbage collection · Middleware

1 Introduction

Cloud computing is a relatively new computing paradigm which allows the virtual sharing of computing resources and yet enables the system management details to remain abstracted from the end users [1, 2]. One of the key enabling technologies for cloud computing is virtualization, which manages the underlying heterogeneous computing resources, abstracts these resources, and provides a uniform interface to the end users. The virtualization technologies in cloud computing consist of two main categories. The first is the language-level virtual machine, e.g. Java Virtual Machine (JVM) [3] and Common Language Runtime (CLR) [4]. These virtual machines provide the "write-once-run-anywhere" capability which allows user applications to run on any type of hardware platforms. The second category is the system-level virtual machine, e.g. VMware [5]. These virtual machines dynamically partition the hardware resources and virtualize the resources for the end users. This allows an m-to-n mapping where m users, each with his/her own operating system image, possibly unaware of each other, can share the n available hardware resources on a strict need basis.

Although virtualization technologies bring many benefits to the cloud computing environment, as virtual machines provide more features, the middleware layer has become bloated, introducing a high overhead. Recent studies [6, 7] have found that the CPU overhead generated by the system virtual machine hypervisor can reach 35% and this number will likely grow in the near future. The situation is similar in language virtual machines: It was found that the garbage collection (GC) module alone consumes on average 10% of the CPU cycles in Apache Harmony [8]. For some data-intensive applications that frequently trigger the garbage collector, the overhead may be even higher.

Our ultimate goal is to provide hardware-assisted solutions to improve the middleware performance in cloud computing environments. By moving some operations to hardware, we expect to reduce the overheads introduced by the middleware layer and leave the computing resources to perform useful work. In detail, we abstract the virtual machine design so that the common software components can be implemented in hardware more cost-efficiently and power-efficiently. As a starting point, we plan to design specialized hardware instructions for GC operations. We select GC because it is a common component in virtual machine designs and it incurs high performance and energy consumption overheads. To this end, we use the Apache Harmony JVM as our test platform to study the behaviors of different GC designs and to provide hardware solutions to accelerate GC.

The contributions of this paper are threefold: first, we identify the common execution "hot spots" during GC; second, we study the GC energy consumption behavior; third, we design, implement, and evaluate hardware-assisted solutions to accelerate

the GC operations. The rest of this paper is organized as follows: in Sect. 2, we review the related work in GC acceleration techniques; in Sect. 3, we introduce various GC algorithms; in Sect. 4, we study the performance of different GC designs and identify the execution "hot spots"; in Sect. 5, we delve into the energy consumption behavior of GC operations; in Sect. 6, we design and implement specialized hardware instructions to accelerate the GC hotspots, and we evaluate the energy consumption and hardware resource utilization of these hardware solutions. In Sect. 7, we conclude and discuss our future work.

2 Related work

Conventional garbage collectors utilize Mark–Sweep algorithms [9, 10] to manage the whole heap. When the heap has no more free space, GC is triggered and the collectors start to trace the heap. If an object can be reached, then it is a live object and the status is marked in the object header or some other metadata area. After tracing, all unmarked objects are swept and their occupied spaces are recycled. The free space in the heap is managed with a linked list. During allocation, the allocators fetch a suitable free region. The main advantage of this algorithm is that no data movement is necessary, such that it incurs low overhead during space recycling. However, Mark–Sweep algorithms introduce heavy fragmentation on the heap, leading to inefficient space utilization. To address this problem, moving GC algorithms are proposed, such as semi-space algorithms and compaction algorithms [11]. Nevertheless, compaction algorithms usually impose lengthy pause time. To reduce pause time, several parallel compaction algorithms have been proposed. Abuaiadh et al. [12] propose a three-phase parallel compactor that uses a block-offset array and mark-bit table to record the live objects moving distance in blocks. Kermany and Petrank [13] propose the Compressor that requires two phases to compact the heap; also Wegiel and Krintz [14] design the Mapping Collector with nearly one phase.

It has been empirically observed that in many programs, the most recently created objects are also those most likely to become unreachable quickly. Generational GCs leverage this property and divides objects into generations [15]. In this case, separate memory regions are used for objects of different generations. For example, the heap can be divided into a Nursery Object Space (NOS) to store newly created objects and a Mature Object Space (MOS) to store mature objects, i.e. the objects that survive one or more collections. When NOS becomes full, GC happens in NOS and moves those few live objects to MOS, and the entire NOS region can then be overwritten with fresh objects; we call this a minor collection.

Special hardware memory modules that support GC have been proposed, the best known of which is the garbage-collected memory module (GCMM) [16] designed for hard real-time applications written in C or C++. Apart from the actual memory devices, the GCMM contains a standard microprocessor and a number of custom circuits, including an arbiter and two elaborate CAM-like devices. Unfortunately, however, the hardware costs of the GCMM are extremely high and in particular prohibitive for embedded systems. Regarding performance, the module's data throughput is considerably inferior to that of standard memory, especially when compared with modern, burst-oriented devices.

Another approach of GC acceleration is the utilization of GC coprocessors. Meyer proposed a garbage collection coprocessor along with the host pipelined *RISC* processor [17]. In this design, GC is executed in micro-codes on the coprocessor. As a result, GC imposes negligible runtime overhead on the host processor. In a subsequent study, the same author went further into a hardware read barrier design, integrating read barrier checking and read barrier fault-handling directly into a processor pipeline [18].

Recently, Joao et al. proposed a hardware-software reference counting approach to reduce the frequency of garbage collection by detecting and reusing dead memory space in hardware [19]. In this approach, hardware reference counters are used, resulting in less frequent garbage collections and consequently less overhead on the overall performance. Nevertheless, this approach is only suitable for garbage collection algorithms that utilize reference counting methods.

Recognizing the importance of the middleware, our ultimate goal is to provide a specialized instruction set to accelerate the middleware layer, and in this paper we focus on GC. Our approach differs from previous work in that we intend to provide a generic GC accelerator instead of focusing on one GC algorithm. To this end, we start by identifying the common "hot spots" in different GC algorithms across a set of benchmarks, and implement a set of hardware instructions to accelerate these common "hot spots." We also study the energy efficiency and hardware utilization of these implementations.

3 GC algorithms

To identify the common "hot spots" in different GC algorithms, we use Apache Harmony as our test platform. We choose Apache Harmony because it is a complete open-source JVM platform that implements many advanced GC algorithms. The default GC in Apache Harmony uses the Partial Forward algorithm (PF) for minor collections, and the Move Compact algorithm (MC) for major collections. But other GC algorithms are available as well, which include the non-generational Mark–Sweep algorithm (MS), the Semi-Space algorithm for minor collections (SS), and the Slide Compact algorithm for major collections (SC), etc. Note that these algorithms are generic and have been implemented in many GC designs. In this section, we introduce the basic concepts behind these algorithms.

As shown in Fig. 1, for better space management in modern GC designs, the heap is divided into multiple spaces. First, the heap is divided into Large Object Space (LOS), which stores objects that are larger than a certain size (usually 4 KB); and regular object space, which stores objects that are less than the size limit. Then based on the property that newly allocated objects are more likely to be recycled, the regular object space is further divided into Nursery Object Space (NOS), which stores newly allocated objects; and Mature Object Space (MOS), which stores objects that have survived one or more garbage collections. Note that by dividing the heap into different spaces, we can apply different GC algorithms in different spaces to achieve maximum GC efficiency. For instance, when NOS is full, we can trigger a GC algorithm to recycle space in NOS only, which we call a minor collection; and when either MOS or LOS is full, we trigger a GC algorithm to recycle space in the whole heap, which

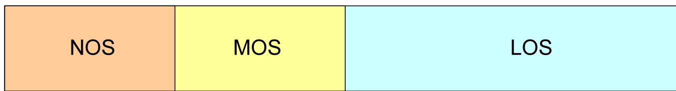

Fig. 1 Generic heap layout in modern Garbage Collection designs

we define as a major collection. On the other hand, we can apply one GC algorithm (such as Mark–Sweep) across the whole heap to minimize GC latency.

3.1 Partial Forward

Partial Forward is one of the collection algorithms for NOS in Apache Harmony. This algorithm leaves the newly allocated objects in NOS and only copies the mature objects (objects that have survived one minor GC to MOS). The assumption is that the newly allocated objects are the most likely to be collected. Hence, we can probably save the copying overheads by leaving the newly allocated objects uncopied until they survive one minor GC. In detail, in Partial Forward algorithm, the old copy of a forwarded object has the FORWARD bit set to one, indicating that the object has been forward to MOS. On the other side, those newly allocated objects have the MARK bit set to one, indicating that the new objects are still alive (not to be recycled). As a result, after a minor collection, a reference to a NOS object can check the MARK bit and the FORWARD bit to decide whether the object has been moved or recycled.

3.2 Semi-Space

The Semi-Space algorithm has been implemented in Apache Harmony GC for NOS collection, or minor collection. When using this algorithm, the NOS is further divided into two spaces: the from-space, where new objects are allocated, and the to-space, which holds the objects that have survived one minor collection. At the beginning of each allocation cycle, programs allocate objects in the from-space. When the from-space is full, a minor collection is triggered, such that it scans the from-space for live objects and moves the live objects to the to-space. Similarly, when the to-space is full, it scans the to-space for live objects and moves the live objects to MOS.

3.3 Move Compact

This algorithm is designed to perform major collections, and as its name implies, it involves object movements. When either LOS or MOS is full, this algorithm first halts all object allocation, and then it scans the heap and marks all the live objects in both MOS and LOS. Next, it starts a process called compaction, in which it moves all live objects toward the left end of both MOS and LOS, thus leaving a large continuous chunk of empty space for future object allocation. By compacting objects toward one end of the heap, compaction algorithms solve the problem of heap fragmentation. Nevertheless, as a trade-off, it takes more time to complete (higher latency) compared to the non-moving algorithms, such as Mark–Sweep.

3.4 Slide Compact

Similarly to the Move Compact algorithm, the Slide Compact algorithm is another compaction algorithm designed for major collections. Typical sliding compaction is done into five phases. In the first phase, it scans the heap to mark the live objects. Then in the second phase, this algorithm computes the new locations for the live objects. In the third phase, it fixes all references that are pointing to the live objects to point to the new locations of these live objects. In the fourth phase, it moves all the live objects to their new locations. And finally, it restores the object header information of these live objects to reflect their location changes.

3.5 Mark–Sweep

On the other hand, Mark–Sweep algorithm does not have to divide the heap into separate spaces, thus both normal and large objects share the same allocation space. The major benefit of Mark–Sweep algorithm is low latency, because it does not have to move objects. When garbage collection is triggered, it scans the heap and marks all live objects. Then it sweeps all the unmarked objects, leaving holes in the heap. These holes are then added into a linked list for future allocations. Although this algorithm is fast, it could introduce serious fragmentation problems in the heap. In cases when fragmentation is very serious, it is unable to find a hole on the linked list to fit a newly allocated object.

4 Performance of Garbage Collection

GC incurs high performance overheads, thus it is worthwhile to provide hardware solutions to accelerate GC. To study the GC performance and identify GC hot spots, we executed various Java benchmark programs on the Apache Harmony JVM and used the Vtune [20] analysis tool to characterize the GC performance. The benchmark programs used in our experiments include six applications from the Dacapo benchmark suite [21], the SPECJVM 2008 benchmark [22], as well as an allocation-intensive program, *Alloc*, taken from [23].

4.1 GC performance overheads

First, we studied the GC execution behavior to characterize the performance overhead it incurs. We used the Vtune analysis tool to capture the fraction of the overall execution time incurred by GC operations. The results are shown in Fig. 2, where the x-axis shows the benchmark names and the y-axis shows the percentage of the overall execution time that is incurred by GC operations in a particular benchmark. The results show that for memory-intensive applications, GC operations incur a high performance overhead. In the execution of *Dacapo-eclipse*, and *Dacapo-xalan*, the GC overhead is over 10 and 15% of the overall execution time, respectively; in *Alloc,* this number even reaches over 20%. When we take an average over all benchmarks, the overhead incurred by GC operations alone is 10.3% of the overall execution time. Considering that GC operations introduce pure management overheads and

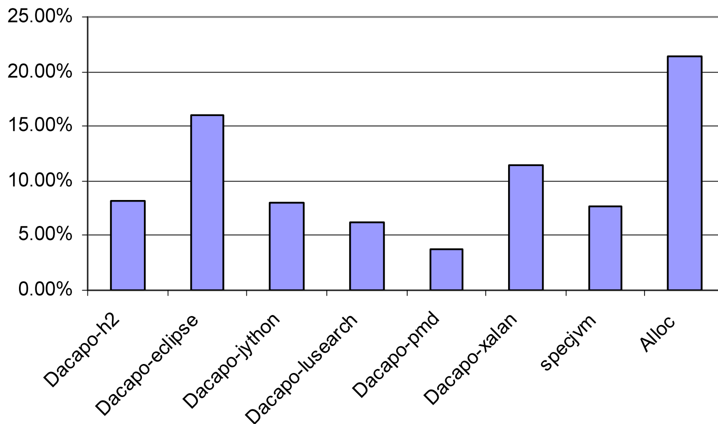

Fig. 2 Fraction of execution time incurred by GC

do not contribute to the execution of the user programs, 10% is a significant overhead. Therefore, these results indicate the need to optimize GC operations and serve as the motivation of this paper.

4.2 Performance hot spots in GC

The GC module in JVM implementations is responsible for memory management and it performs two main tasks: object allocation and garbage collection. During program execution, where there is an object allocation request, the GC fetches a chunk of free memory from the heap and returns. If not enough space is in the heap to satisfy the allocation request, then garbage collection is triggered, such that the GC scans the heap to recycle the dead objects.

In this subsection we study the GC execution behavior in order to identify the execution hot spots. By identifying these hot spots, we can design the hardware accelerators to speed up GC execution. In the first experiment, we executed the benchmark programs on Apache Harmony using the default GC. We observed that the top three hot spots of the GC executions were the functions *gc_alloc_fast*, *scan_slot*, and the *vtable* manipulation functions, and this observation was consistent across all benchmarks. Table 1 summarizes the results: the first column lists all benchmark programs; the columns, *%alloc_fast*, *%vt*, and *%scan_slot*, respectively, show the fraction of GC execution time incurred by the functions *gc_alloc_fast*, *vtable* manipulations, and *scan_slot*; the last column, *%total* shows the fraction of GC execution time incurred by the combination of these functions. The last row of Table 1 shows the average results across all benchmarks. The results indicate that on average, these functions contribute to almost 50% of the GC execution time. In the case of *xalan*, a data-intensive application, the *gc_alloc_fast* alone contributes to almost 50% of the GC execution time, and the combination of these functions contributes almost 70% of the GC execution time.

In the second experiment, we executed the *xalan* benchmark on Apache Harmony using different garbage collection algorithms. We chose *xalan* because it is a data-intensive application, which will stress the GC module in the Apache Harmony JVM.

Table 1 Hot spots in the default GC

Benchmark	%alloc_fast	%vt	%scan_slot	%total
h2	7	23.5	11	41.5
Eclipse	17	22	10	49
Jython	24.5	20	8	52.5
Lusearch	21	18	8	47
Pmd	13.5	21	6	40.5
Xalan	46	17	6	69
Specjvm	20.5	19	5.5	45
Alloc	3	30	5	38
Average	19.1	21.3	7.4	47.8

Table 2 Hot spots in various GC algorithms

Algorithm	%alloc_fast	%vt	%scan_slot	%total
PF–SC	45	14	4	63
PF–MC	46	17	6	69
SS–MC	21	13.5	3.5	38
SS–SC	17	16	6	39
MS	27	5	2	34
Average	31.2	13.1	4.3	48.6

In our observation, the top three hot spots of the GC executions were still the functions *gc_alloc_fast*, *scan_slot*, and the *vtable* manipulation functions. Table 2 summarizes the results: the first column lists the GC algorithms: for example, *PF–SC* indicates that the Partial Forward algorithm was used for minor collections and the Slide Compact algorithm was used for major collections; and *SS–MC* indicates that the Semi-Space algorithm was used for minor collections and the Move Compact algorithm was used for major collections, etc. Note that *MS* indicates that the Mark–Sweep algorithm was used for the whole heap collection, and no generational algorithm was used. The rest of the table organization is the same as in Table 1. The results are similar to those in the previous experiment, such that on average, these functions contribute to almost 50% of the GC execution time. By accelerating these functions, we expect to reduce the execution overhead incurred by the GC operations.

4.3 Analysis of the GC hot spots

Before designing hardware accelerators for these GC hot spots, we analyze the details of these functions. Figure 3 illustrates the pseudo-codes of the function *gc_alloc_fast*. In a straightforward allocator design, whenever there is an object allocation request, the allocator thread attempts to obtain the requested free space from the heap. However, the heap is a shared resource and multiple threads may be contending to access it, which dramatically slows down the operation. To address this problem, each thread can reserve a block of the global memory, which is termed *thread local storage*, or *TLS*. In this case, the allocator thread does not have to contend for heap access; instead, it just allocates from its own TLS with so-called “bump-pointer” allocation, and

```
1.    object gc_alloc_fast (int size){
2.      // step 1: get thread local storage
3.      void* tls_base = vm_thread_local();
4.      void* tls_addr = tls_base + tls_gc_offset;
5.      Allocator* allocator = (Allocator*)*(tls_addr);
6.
7.      //step 2: check object size
8.      if ( size > GC_LOS_OBJ_SIZE_THRESHOLD )
9.          return NULL;
10.
11.     //step 3: try to allocate object
12.     int free = allocator->free;
13.     int ceiling = allocator->ceiling;
14.     int new_free = free + size;
15.
16.     if (new_free <= ceiling)
17.       allocator->free= (void*)new_free;
18.       return free;
19.     else
20.      return NULL;
21.   }
```

Fig. 3 Pseudo-codes of *gc_alloc_fast*

Fig. 4 Pseudo-codes of *vtable* manipulations

```
1.    GC_VTable_Info *vtable_get_gcvt(VT* vt)
2.    {
3.      GC_Vtable_Info* gcvt = vt->gcvt
4.      return gcvt & GC_CLASS_FLAGS_MASK;
5.    }
6.
7.    GC_VTable_Info *obj_get_gcvt(Object *obj)
8.    {
9.      VT* vtable = obj->vt_raw;
10.     vtable = vtable & ~CLEAR_VT_MARK;
11.     return vtable_get_gcvt(vtable);
12.   }
13.
14.   Boolean obj_mark_in_vt(Object *obj)
15.   {
16.     VT vt = obj_get_vt_raw(obj);
17.     if((vt & CONST_MARK_BIT)
18.         return FALSE;
19.     vt = vt | CONST_MARK_BIT;
20.     obj_set_vt(obj,  vt );
21.     return TRUE;
22.   }
```

thus largely speeds up the operation. The function *gc_alloc_fast* performs exactly this operation: in the first step, the address of the TLS is identified; in the second step, it checks whether the object size is too big to allocate in the TLS; and in the final step, it checks whether the TLS contains enough space for the allocation; and if so, the object is allocated.

Figure 4 shows several *vtable* manipulation functions. In JVM implementations, each object contains a header, which records the attributes and information of the object; *vtable*, or virtual method table, is a part of the object header. In Apache Harmony implementation, the *vtable* structure contains the virtual table of the object methods. The VM reserves 4 bytes for exclusive use by the GC. The GC uses 4 bytes of GC-private space to put the pointer to the object information structure. Many operations on the objects require the manipulation of the related information stored in *vtable*. For example, when a new object is created, its *vtable* pointer needs

```
1.    void scan_slot(Collector* collector, REF *p_ref)
2.    {
3.       //step 1: get object from reference
4.       object p_obj = read_slot(p_ref);
5.       if( p_obj == NULL) return;
6.
7.       //step 2: return if object has already been marked
8.       if(!obj_mark_in_vt(p_obj))
9.           return;
10.
11.      //step 3: push object to the trace stack
12.      Vector_Block* trace_task;
13.      trace_task = (Vector_Block*)collector->trace_stack;
14.      trace_task->head--;
15.      *(trace_task->head) = p_obj;
16.
17.      //step 4: get a new trace stack if the current one is full
18.      sync_stack_push(mark_task_pool, trace_task)
19.      collector->trace_stack = sync_stack_pop(free_task_pool);
20.   }
```

Fig. 5 Pseudo-codes of *scan_slot*

to be set in the object header; also, when GC occurs, the *vtable* fields in the object header need to be marked to indicate that the object is reachable. The *vtable* manipulation functions perform these functions and contribute to a large portion of the GC execution time. Figure 4 shows three of these functions: *vtable_get_gcvt* retrieves the GC *vtable* information and masks it with the *GC_CLASS_FLAGS_MASK*; *obj_get_gcvt* takes an object pointer as input and retrieves its GC *vtable* information, note that *vtable_get_gcvt* is called within this function; *obj_mark_in_vt* marks an object's *vtable* to indicate that the object is alive. Note that the *vtable* manipulation functions also include *obj_get_vt_raw*, *obj_get_vt*, *obj_set_vt*, *vtable_get_gcvt_raw*, and *obj_get_gcvt_raw*.

Figure 5 illustrates the pseudo-codes of the function *scan_slot*. When GC is triggered to recycle dead objects, it scans the references in each reachable object recursively from the root set, and checks whether the reference points to a reachable object that has not yet been discovered. The function *scan_slot* performs exactly this operation: in the first step, it checks whether the reference points to an object; in the second step, it checks whether the object has already been marked and it marks the object if the object has never been marked; in the third step, it pushes the object to the trace stack, such that later it can trace the references contained in this object; and in the final step, it checks whether the current trace stack is full, and if so, it obtains another trace stack.

In order to identify the execution time of each of these hot spots, we used Vtune to capture the number of instructions and the clocks per instruction measure for each function; then we multiplied these two numbers to derive the number of cycles taken to execute each function. The results show that *gc_alloc_fast* takes 120 cycles to complete, *scan_slot* takes 105 cycles to complete, and on average the *vtable* manipulation functions take 20 cycles to complete.

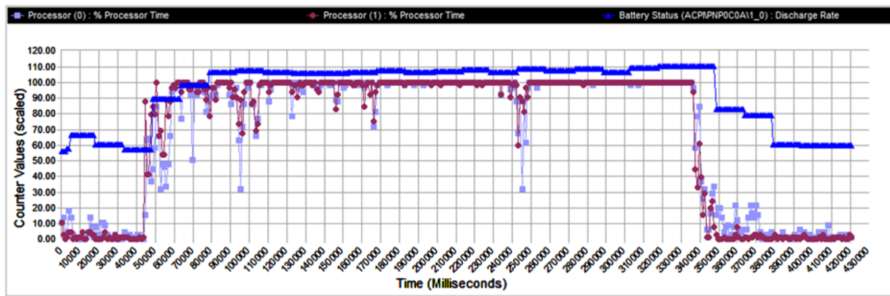

Fig. 6 Energy consumption of the Apache Harmony JVM execution

5 Energy consumption of GC

In this section, we estimate the energy consumption incurred by each invocation of the GC hotspot functions. To achieve this, we follow the techniques presented in [24] such that we combine real-time total power measurements with performance-counter-based per-unit power estimation. In detail, live power measurements for the running system are obtained from the Vtune battery discharge rate monitor. In parallel, the processor hardware performance counter readings are obtained to derive the energy consumption information for each function. Then we combined these two pieces of information to derive energy consumption of the execution of individual software components.

First, we studied how the Apache Harmony JVM execution affects the overall system energy consumption. We selected the default GC and executed *xalan*. Then, we utilized Vtune to monitor the power consumption as well as processor activity during the Apache Harmony JVM execution. The real-time measurement results are shown in Fig. 6. The y-axis shows the normalized processor activity and battery discharge rate, and the x-axis shows execution time in milliseconds. The red dotted line and the gray square line represent the activity of the two CPU cores; whereas the blue triangle line represents the battery discharge rate. At the beginning, both the processor activity and the battery discharge rate are low due to the lack of activity in the system. Once the JVM execution starts, the activities of both processor cores increase to almost 100% and the system starts drawing more power. The processor activities and system power consumption stay high for about five minutes until the execution completes. Then the processor switches into idle mode; and the power management module detects the processor inactivity and reduces the battery discharge rate to conserve energy. The energy consumption for each JVM execution can then be calculated by taking the product of the voltage, the discharge current, and the execution time. Using the average value of five executions, we calculated that each execution of *xalan* on Apache Harmony consumed 213 joules of energy.

Second, we estimated the energy consumption of each invocation of the GC hotspot functions. While there is no direct way to measure this data, we utilized the counter-based power consumption approximation approach used in [24]. The counter-based energy consumption model we used is presented in (1), which approximates the total system energy consumption E as the summation of energy consumed

Table 3 Energy consumption of GC hot spots

Hot spots	total E (J)	count (10^6)	E (nJ)
gc_alloc_fast	3.515	100	17.6
scan_slot	0.208	0.84	124
obj_get_vt	0.384	40	4.8
vtable_get_gcvt_raw	1.329	121	5.5
obj_get_gcvt_raw	0.109	15	3.6
vtable_get_gcvt	0.180	25	3.5
obj_get_gcvt	0.217	25	4.2

by hardware components, including the processors, the system bus, and the memory modules. The energy consumed by component i can be approximated by the product of the number of times the component is triggered, C_i, and the energy consumed by the component each time, E_i. To approximate the energy consumption of the processors, we utilized the hardware performance counter to obtain the number of retired instructions; for the system bus energy consumption, we obtained the number of bus transactions during the JVM execution; and for the memory subsystem energy consumption, we monitored the number of memory accesses during the JVM execution.

$$E = \sum_{i=0}^{n} C_i E_i \tag{1}$$

Next, to find out the energy consumption for each GC hotspot functions, we used Vtune to capture the number of retired instructions, the number of bus transactions, and the number of memory accesses incurred by each of the GC hotspot functions. By taking all this counter information into (1) and assuming that each activity consumes the same amount of energy (e.g. E_i is the same for memory access, bus transaction, and instruction execution), we were able to approximate the energy consumption incurred by each of the GC hotspot functions and the results are summarized in Table 3. The second column shows the total energy consumption incurred by all invocations of a hotspot function; the third column shows the number of invocations during execution; and the last column shows the average energy consumption incurred by each invocation. The results indicate that each invocation of *gc_alloc_fast* consumes 17.6 nJ of energy, each invocation of *scan_slot* consumes 124 nJ of energy. Also, on average, each *vtable* manipulation function consumes 4.3 nJ of energy.

6 Hardware-assisted Garbage Collection

In this section, we explore hardware solutions to accelerate the GC hotspots. We implement the design and evaluate the performance, hardware utilization, and energy consumption of the hardware solution.

6.1 System integration

The traditional method to incorporate an accelerator into a system is by designing a special hardware module (e.g. [16]) or a coprocessor (e.g. [17]). This method does

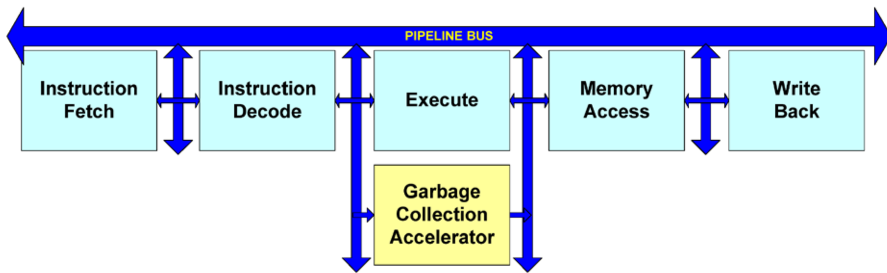

Fig. 7 System integration of the GC accelerator

not require the modification of existing hardware; nevertheless, this approach suffers from several disadvantages. The first one is the lack of flexibility: by having a dedicated hardware module, the module can target only one GC algorithm. Thus, each time there is a change of GC algorithm or modification of existing GC algorithms, the hardware module needs to be changed or modified. However, it is very difficult and expensive to re-design and re-implement the hardware modules. Second, by having a dedicated module, the communication overhead can be as high as tens of cycles (if the module stays in the same chip as the host processor), or even hundreds of cycles (if the module stays in a different chip). This high communication overhead may offset the performance benefits brought by the accelerator modules.

On the other hand, instead of accelerating specific GC algorithms on dedicated hardware modules, we target basic operations (those identified in Sect. 4) that are generic across different GC algorithms. As illustrated in Fig. 7, for GC acceleration, our plan is to extend the Execution Logic (ALU) in existing pipelines to incorporate hardware accelerators of these basic GC operations. This design brings two benefits: first, even if the GC algorithm has been changed, it is very likely that the new GC algorithm still uses all, or at least part of, the same set of basic operations, thus the accelerator would still be able to accelerate the GC operations. Second, by tightly coupling the accelerator into the host pipeline, the communication overhead between the host pipeline and the accelerator becomes negligible, thus maximizing the benefits of the GC accelerator.

For simplicity, we use a five-stage MIPS pipeline as an example in Fig. 7. At the beginning, the Instruction Fetch unit fetches a new instruction from the instruction memory; this instruction can be the legacy MIPS instruction or the proposed GC operations. Then the Instruction Decoder decodes the instruction and decides whether the ALU or the Garbage Collection Accelerator should execute this instruction. Then the rest of the pipeline remains the same, after going through the Memory Access and Write Back stages, the instruction retires.

6.2 Design of the Garbage Collection Accelerator

Now we look into the design of the Garbage Collection Accelerator (GCA). Recall that in Sect. 4 we have identified three sets of the basic GC operations, including *gc_alloc_fast*, *scan_slot*, and *vtable_ops*. As shown in Fig. 8, our GCA design incorporates hardware accelerators for these basic GC operations and data flows from

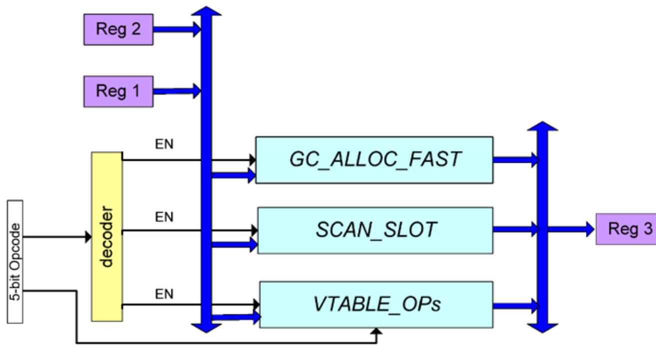

Fig. 8 Design of the GC accelerator module

left to right in the GCA. At the beginning, a five-bit *Opcode* is required to choose the property GC operation; also, *Reg1* and *Reg2* store the input values, e.g. the size of allocation in *gc_alloc_fast*. Then, based on the five-bit *Opcode*, the *decoder* enables one of the hardware modules to performance computation. After the operation is complete, the result or output is stored in *Reg3* such that the host pipeline can retrieve it.

To implement the basic GC operations in hardware, we apply two optimizations: configuration registers and parallel execution. Before JVM execution starts, the configuration parameters, such as *tls_gc_offset* (line 4 in Fig. 3), are set in the program. These configuration parameters are frequently used in the execution. However, in the software implementation, each time these configuration parameters are accessed, they need to be loaded from memory into registers. This incurs high performance and power consumption overhead.

To address this problem, we utilized dedicated configuration registers to store these configuration parameters in the hardware implementation. For instance, in the *vtable* manipulation functions, the offsets *vt_raw* and *gcvt* can be pre-loaded into the configuration registers. As a result, during the execution of the simple function *obj_get_gcvt*, instead of issuing two memory-access instructions to load the values of *vt_raw* and *gcvt*, these values can be obtained from the configuration registers. In this way, all the computations can be completed in one instruction instead of spreading over multiple instructions.

To identify opportunities for parallel execution, we generated the dataflow graphs of these hotspot functions and converted them into finite state machines. For instance, in *gc_alloc_fast*, when there is an allocation request, it checks whether the allocation size is valid; also, it needs to identify the starting address of the thread local storage. Although these two operations are arranged in sequential order in the software implementation, there are no data dependencies between these operations according to the dataflow graph. Hence, we can execute these two operations in parallel and abstract them into one state in the finite state machine. Following this methodology, the designs of the hardware implementations are as follows.

gc_alloc_fast

In the first state, the input values are loaded into the registers and the address of the thread local storage is identified. In the second state, it checks whether the size of the allocation request is valid, and also it checks the base and ceiling address of the thread local storage. In the third state, it computes the new ceiling address of the thread local storage. In the fourth state, it checks whether the thread local storage contains sufficient space for the allocation. In the fifth state, it updates the base address of the thread local storage and generates address for the allocation. In the sixth state, the address is returned and the operation completes.

scan_slot

In the first state, the input values are loaded into registers and it checks whether the input reference is valid. In the second state, it gets the *vtable* value from the object information. In the third state, it obtains the address of the trace stack. In the fourth state, it pushes the object into the trace stack and checks whether the trace stack is full. In the fifth state, it saves the current trace stack. In the final state, it obtains a new trace stack for further scanning operations.

vtable manipulations

There are multiple *vtable* manipulation functions, and these functions are all implemented in the same hardware unit since there are a lot of operations that overlap with each other. Two functions, *obj_get_gcvt_raw* and *obj_get_gcvt*, take three cycles to complete and they are the critical path of this hardware unit. Other functions either take two cycles or only one cycle to complete; these shorter functions may be included in the longer functions. For example, the function *obj_get_gcvt_raw* calls the function *vtable_get_gcvt_raw*, and *vtable_get_gcvt_raw* takes two cycles to complete. Most of these functions involve the loading or modification of the object header information. In the case of *obj_get_gcvt_raw*, in the first state, it loads the input into the address register. In the second state, it obtains the *gcvt* value of the object header. In the third state, it loads the *gcvt* value into the output register and returns.

6.3 Implementation and evaluation

We implemented the hardware-assisted GC functions on the Xilinx Spartan III FPGA board [25]. We also measured the hardware resource utilization, performance, and energy consumption. Table 4 summarizes the hardware resource utilization of these implementations: the hardware utilization measures include the number of hardware slices (*#slices*), the number flip-flops (*#FF*), and the number of look-up tables (*#LUT*). As a comparison, we also present the resource utilization of a simple MIPS

Table 4 Hardware resource utilization

	#slices	*#FF*	*#LUT*
gc_alloc_fast	170	180	220
scan_slot	140	140	270
vtable	120	110	80

Table 5 Performance and energy consumption

Function	time (cycles)	E (nJ)
gc_alloc_fast	6	9.6
scan_slot	6	10.8
obj_set_vt	1	1.6
vtable_set_gcvt	1	1.6
get_obj_info_raw	2	3.2
obj_get_vt_raw	2	3.2
obj_get_vt	2	3.2
vtable_get_gcvt_raw	2	3.2
obj_get_gcvt_raw	3	4.8
obj_get_gcvt	3	4.8

in-order processor [28], which requires 10,450 hardware slices, 10,400 flip-flops, and 19,500 look-up tables. Thus, the hardware-assisted *gc_alloc_fast*, *scan_slot*, and *vtable* functions incur about 1.5, 1.5, and 1% respectively of hardware resource overheads compared to the simple in-order MIPS processor.

Table 5 summarizes the latency and energy of each invocation of these hardware implementations. The second column lists the number of cycles taken to execute these functions. The third column lists the energy consumption for each invocation, which is derived by multiplying the execution time by the power consumption of the design. We obtained the power consumption information by using the Xilinx XPower [26]: the power consumption for *gc_alloc_fast, scan_slot,* and *vtable* manipulations is respectively 0.8, 0.9, and 0.8 W. Compared to the software implementation, the hardware *gc_alloc_fast* results in 20-fold performance improvement and 2-fold energy consumption improvement; the hardware *scan_slot* results in 18-fold performance improvement and 12-fold energy consumption improvement; and on average, the *vtable* manipulation functions take 2 cycles to complete and consume 3.2 nJ of energy, representing a 10-fold performance improvement and 50% energy consumption improvement. Note that this comparison is based on the FPGA technology. According to [27], the ASIC technology is about 12 times more power-efficient than the FPGA technology. Therefore, by moving these hardware implementations to ASIC designs, we expect to get better energy efficiency.

6.4 Performance estimation

We have demonstrated that the hardware implementations make possible to accelerate the GC hotspots *gc_alloc_fast*, *scan_slot*, and *vtable* manipulation functions by 20, 18, and 10 times, respectively. To estimate how the hardware implementations would improve performance when integrated into a system, we utilized the speedup data and conducted an extrapolation study. From Table 2, we obtained the workloads of these three hotspot functions relative to the total GC time; then we applied the speedup information of these functions and calculated how the total GC time would be affected. Recall that the results in Table 2 were gathered using the *xalan* benchmark, and according to Fig. 1 the GC time contributes to 11.2% of the total execution

Table 6 Performance estimation with GC acceleration

Algorithm	%GC	GC time	Overall time
PF–SC	63	0.42	0.93
PF–MC	69	0.42	0.93
SS–MC	38	0.40	0.93
SS–SC	39	0.41	0.93
Unique MS	34	0.39	0.93
Average	48.6	0.41	0.93

time in *xalan*, so we were able to derive how the hardware implementations affect the total JVM execution time. The results of our performance estimation study are summarized in Table 6: the second column lists the percentage of the total GC execution contributed by these three hotspot functions; the third column lists the fraction of the new GC time (by using the hardware implementations) relative to the original GC time (by using pure software implementations); and the last column lists the fraction of the new JVM execution time relative to the original JVM execution time. The results indicate that on average these three hardware implementations make possible to reduce the total GC time by 60% on various GC algorithms, and this GC performance improvement contributes to 7% reduction of the total JVM execution time.

7 Conclusions

In this paper, we designed, implemented, and evaluated specialized hardware instructions to accelerate GC operations. We selected GC because it is a common component in virtual machine designs and it incurs high performance and energy consumption overheads.

Our profiling results indicate that GC incurs significant overheads, contributing to about 10% of the total execution time. By examining the details of GC execution using different GC algorithms, we identified three performance hotspots. These hotspots contribute to more than 50% of the total GC execution time, and they also consume a significant amount of energy. By moving these hotspot functions into hardware, we achieved an order of magnitude speedup and significant improvement on energy efficiency. In addition, we also performed an extrapolation study by using the hardware parameters into the profiling data. The results indicate that these three simple hardware-assisted GC instructions can reduce the GC execution time by more than half and lead to a 7% improvement on the overall execution time.

Note that our ultimate goal is to provide hardware-assisted solutions to improve the middleware performance in cloud computing environments. Thus in the next step we plan to provide hardware solutions to accelerate other parts of system and language virtual machines. Afterwards, we plan to incorporate these specialized instructions into an architectural simulator and perform system-level study to evaluate how these specialized instructions can improve the performance of the middleware layer.

Acknowledgements This work is partly supported by Natural Science Fund of China (No. 61070029) and by the National Science Foundation under Grant No. CCF-0541403. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

References

1. Armbrust M, Fox A, Griffith R, Joseph A, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zararia M (2009) Above the clouds: a Berkeley view of cloud computing. Technical Report No. UCB/EECS-2009-28, University of California, Berkeley, USA
2. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I (2009) Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener Comput Syst 25:599–616
3. Apache Harmony http://harmony.apache.org/
4. Common Language Runtime http://msdn.microsoft.com/en-us/library/8bs2ecf4(VS.71).aspx
5. VMware http://www.vmware.com/
6. Cherkasova L, Gardner R (2005) Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In Proceedings of the annual conference on USENIX annual technical conference
7. Ferrer M Measuring overhead introduced by VMWare workstation hosted virtual machine monitor network subsystem. http://studies.ac.upc.edu/doctorat/ENGRAP/Miquel.pdf
8. Liu S, Wang L, Li X-F, Gaudiot J-L (2009) Packer an innovative space-time efficient parallel garbage collection algorithm based on virtual spaces. In Proceedings of the 24th IEEE international parallel and distributed processing symposium
9. Toshio E, Taura K, Yonezawa A (1997) A scalable Mark–Sweep garbage collector on large-scale shared-memory machines. In Proceedings of ACM/IEEE conference on supercomputing, New York, USA, 1997
10. Domani T, Kolodner EK, Lewis E et al (2001) Implementing an on-the-fly garbage collector for Java. ACM SIGPLAN Notices
11. Jones RE (1996) Garbage collection: algorithms for automatic dynamic memory management. Wiley, Chichester. With a chapter on distributed garbage collection by R Lins
12. Abuaiadh D, Ossia Y, Petrank E, Silbershtein U (2004) An efficient parallel heap compaction algorithm. In the ACM conference on object-oriented systems, languages and applications
13. Kermany H, Petrank E (2006) The compressor: concurrent, incremental and parallel compaction. In PLDI
14. Wegiel M, Krintz C (2008) The mapping collector: virtual memory support for generational, parallel, and concurrent compaction. In ASPLOS ’08, Seattle, WA
15. Appel W (1989) Simple generational garbage collection and fast allocation. Softw Pract Exp 19:171–183
16. Schmidt WJ, Nilsen KD (1994) Performance of a hardware-assisted real-time garbage collector. In International conference on architectural support for programming languages and operating systems, pp 76–85
17. Meyer M (2004) A novel processor architecture with exact tag-free pointers. IEEE MICRO 24(3):46–55
18. Meyer M (2006) A true hardware read barrier. In ISMM’06
19. Joao JA, Mutlu O, Patt YN (2009) Flexible reference-counting-based hardware acceleration for garbage collection. In Proceedings of the 36th annual international symposium on computer architecture
20. Intel Vtune http://software.intel.com/en-us/intel-vtune/
21. Dacapo Project: The DaCapo Benchmark Suite http://www-ali.cs.umass.edu/dacapo/index.html
22. SPECJVM 2008 http://www.spec.org/jvm2008/
23. Liu S, Deng C, Li X-F, Gaudiot J-L (2009) RHE: a lightweight JVM instructional tool. In Proceedings of the 33rd annual IEEE international computer software and applications conference

24. Isci C, Martonosi M (2003) Runtime power monitoring in high-end processors: methodology and empirical data. In Proceedings of the 36th international symposium on microarchitecture
25. Xilinx Spartan III http://www.xilinx.com/products/devkits/HW-SPAR3-SK-UNI-G.htm
26. Xilinx XPower http://www.xilinx.com/products/design_tools/logic_design/verification/xpower.htm
27. Kuon I, Rose J (2006) Measuring the gap between FPGAs and ASICs. In Proceedings of the 14th international symposium on field programmable gate arrays
28. The eMIPS project http://research.microsoft.com/en-us/projects/emips/default.aspx