Garbage Collector Interface

Anders Ive anders.ive@cs.lth.se
Anders Blomdell anders.blomdell@control.lth.se
Torbjörn Ekman torbjorn.ekman@cs.lth.se
Roger Henriksson roger.henriksson@cs.lth.se
Anders Nilsson anders.nilsson@cs.lth.se
Klas Nilsson klas.nilsson@cs.lth.se
Sven Robertz–Gestegård sven@cs.lth.se

Abstract

The purpose of the presented garbage collector interface is to provide a universal interface for many different implementations of garbage collectors. This is to simplify integration and exchange of garbage collectors, but also to support incremental, non–conservative, and thread safe implementations. Due to the complexity of the interface, it is aimed at code generators and preprocessors. Experiences from ongoing implementations indicate that the garbage collector interface successfully provides the necessary functionality in an efficient way.

1 Introduction

The use of automatic memory management, usually referred to as garbage collection carried out by a garbage collector, *GC*, has proven to improve productivity and quality in software development. Productivity increases up to 200% have been reported, and the quality improvement (without additional extensive testing) is significant when programming is performed using a safe language. By a safe language we mean that all possible executions are expressed by the program itself, which in turn requires automatic management of memory (that is, a GC).

When it comes to real–time systems, there is a widely spread misconception that the presence of a GC results in unpredictable timing and too large memory footprint. In our earlier work we have shown that efficient real–time GC, *RTGC*, is possible, with some assumptions on compilers and run–time system properties [Hen98]. However, for the more general case with a fully preemptive natively multithreaded run–time system, and considering a desire to interface code written in a safe language (like Java) with external code written (or created via code generation tools) in a standard language such as C, there are (to our knowledge) no published results or available products that provides RTGC. To make a modern language like Java useful in the embedded world it is important to enable clear and complete access to the workings of the real-time garbage collector.

The garbage collector interface, *GCI*, enables many different garbage collector algorithms to be interchanged in an application, e.g. thread safe, incremental, and non–conservative implementations. Currently, many systems with support for automatic memory management suffer from the penalties of old

and conservative implementations. For example, the popular GNU C–compiler, gcc, supports an old and conservative "stop–the–world" garbage collector (see [gcc00]) invented by Boehm and Weiser, [BW88]. Benefits from other garbage collectors may not directly easily be supported. However, some applications do not need the advantages of modern garbage collectors. The garbage collector interface covers these algorithms as well.

The interface will support common garbage collecting algorithms in existence today, and hopefully be extensive enough to cover future garbage collector implementations. The GCI supports multithreaded systems. Future work will be aimed to cover multiprocessor systems, networks, arrays, matrices and garbage collector configuration. Our implementation of GCI is written in C.

Currently, the GCI is complex and mainly targeted towards code generators and preprocessors. To enable garbage collection to the programmer in a simple way, the compiler has to be modified.

1.1 Garbage Collection

As soon as manual memory management is introduced to an application, a GC is often desirable to avoid nasty bugs such as dangling pointers or pessimistic memory deallocation that result in low utilisation of the memory. Many applications allocate more memory than is deallocated. They may run out of memory if deallocation is not performed correctly. Inevitable crashes and low performance are imminent if those problems are not solved.

Automatic memory management relieves the programmer from the error prone deallocation of memory. Optimally, memory should be deallocated when it will never be utilised again. However, contemporary garbage collector algorithms assume that allocated memory areas, *objects*, may be deallocated when it is impossible to access the objects. That condition is true when there are no references[1] to the object. The garbage collector must analyse every *live* reference to draw the conclusion that there are no references to an object. The complete memory area where every object resides is called the *heap*.

The locations of the references in an object have to be described. The description is often stored in the *layout* of the object, together with information that is common to all the objects, *instances*, created from the layout, e.g. the object size. One alternative is to store the location of the references and the object size inside the instance itself at the expense of larger instances. However, the performance may increase as the access to elements inside objects can be made faster.

The following sections describe how the requirements of GC algorithms, and multithreaded systems are reflected in the GCI. Table 1 shows the different layers of the GCI. Code examples show how the GCI will influence memory management, first from the simpler view of the garbage collector, and then from the more complex multithreaded perspective (which is also the utilised interface). The specification of the GCI is described in Section 4. Results from utilisation of the GCI are covered in Section 5 followed by conclusions that summarises the article.

[1] The difference between a pointer and a reference is that the reference may only refer to objects while pointers may point to anything, e.g. an integer value or even outside the scope of the memory.

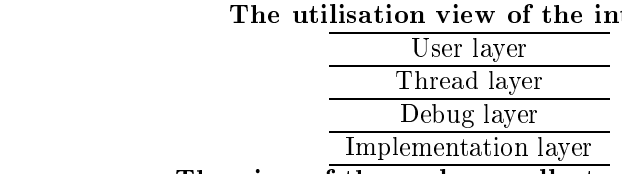

Figure 1: The garbage collector interface is structured as layers even though it is only accessible from the user layer.

2 Interface to the Garbage Collector

The lowest layer of the GCI handles references. It is through the references that objects can be utilised. It is crucial for a GC to maintain the references correctly. This section describes the situations where reference handling occurs in an application.

2.1 References

It is essential that the garbage collector keeps track on all references in an application at all time. Otherwise, the GC cannot guarantee that the memory management is performed correctly.

To keep track of all references, their declaration, utilisation, and termination must be informed to the garbage collector. The implementation layer of the GCI mainly targets the supervision of references in an algorithm independent way. All statements that handle references are covered by C–macros. Figure 2, Figure 3, and Figure 4 shows a small C–program that utilises manual memory management and the equivalent program written with consideration to the implementation layer of the GCI.

The example presumes that preemption does not occur arbitrarily within the code. Still, preemption could be supported at specific and safe *preemption points* that are manually inserted. During preemption, the GC has to have complete control over the locations of all the references in the application, i.e. the *reference graph* has to be consistent.

The reference graph spans all the reachable references, and objects, in the application. The entry references of the graph are called *roots*, and they are stored in a *root set*. All the objects that can be reached from the roots directly or indirectly via references inside objects, are alive since they may be utilised by the application. The other non–reachable objects cannot be utilised from the application. They are reclaimed.

It is important to keep the reference graph consistent where it may be accessed from the code, to circumvent unpredictable and erroneous program behaviour. The modifications of the reference graph must be performed undisturbed. The program that utilises objects, i.e. the *mutator*, and the GC mustn't modify the graph at the same time to avoid inconsistencies of the live references. The *write barrier* prevents the mutator to mutate the reference graph so it interferes with the workings of the GC. The *read barrier* hinders the mutator to access references that are not up–to–date. Preemption is treated in Section 3, where the thread layer of the GCI is described.

There exist *conservative* garbage collectors that loosen the requirement of control over references. In conservative GC, every number that resembles a reference is treated as such; an explicit reference graph is omitted. The unpredictable and unreliable behaviour of conservative garbage collectors renders them unsuitable for real–time applications. Conservative GC are not considered further in this paper since the usage of them may result in, for example, memory leaks. However, even the conservative GC can be completely covered by the GCI. In fact, only a subset of the GCI is required to fully cover the conservative approach.

```
typedef struct {
  struct Object* o1;
  int i1, i2;
  struct Object* o2;
  struct Object* o3;
  int i3;
} Object;
```

```
typedef GC_STRUCT_BEGIN(Object)
  GC_STRUCT_REF(Object o1);
  GC_STRUCT_VAR(int, i1);
  GC_STRUCT_VAR(int, i2);
  GC_STRUCT_REF(Object o2);
  GC_STRUCT_REF(Object o3);
  GC_STRUCT_VAR(int, i3);
GC_STRUCT_END(Object) Object;
```

Figure 2: A C–structure is converted to an object description in the garbage collector interface.

2.2 Implementation Layer Utilisation

The implementation layer of the garbage collector interface focuses on supervising the references in a program. All garbage collectors require support for reference handling. Our approach with the GCI is to describe C–macros that hide the reference utilisation. The following subsections describe the types of reference handling in GC. The macros that are described in the subsections are utilised in the code examples in Figure 2 (object declaration), Figure 3 (global reference), and in Figure 4 (main program).

This section handles single threaded applications. Consideration about preemption in multithreaded systems is studied in detail in Section 3.

The macros of the GCI can be divided into different categories due to the functionality. The categories and the macros of the GCI is shown in Table 1. The categories *Read barrier* and *Write barrier* are related to common GC characteristics (copying and incremental). A detailed description of various garbage collector algorithms is covered in the "Garbage Collection" book by Richard Jones and Rafael Lins, [JL96]. The following subsections describe the various categories:

```
struct Object* saved;
```

```
GC_STATIC_REF(Object, saved);
```

Figure 3: The global variable `saved` is declared as a reference to an object of type `Object`.

```
int main() {
  Object* r1;
  Object* r2;
  Object* r3;

  // Allocate objects
  r2 = (Object*) malloc(sizeof(Object));
  r3 = (Object*) malloc(sizeof(Object));

  // Access objects
  r1 = r2;
  r3->ref1 = r2;

  // Deallocate objects
  free(r2);
  free(r3);

  return 0;
}
```

```
int main() {
  // Initialisation
  GC_INIT(10000);         // Initialise GC and heap
  GC_STATIC_BEGIN(saved); // Register global reference as root

  // Register the local references as roots
  GC_ROOT_BEGIN(Object, r1);
  GC_ROOT_BEGIN(Object, r2);
  GC_ROOT_BEGIN(Object, r3);
  GC_ROOT_BEGIN(Layout, layout); // Layout is an internal
 // structure describing layouts

  // Create layout for Object
  GC_NEW_LAYOUT(layout, Object, 3,
      GC_LAYOUT_FIELD(Object, o1),
      GC_LAYOUT_FIELD(Object, o2),
      GC_LAYOUT_FIELD(Object, o3));

  // Instantiate new object from the 'layout'
  GC_NEW(ref, layout);

  // Allocate objects
  GC_NEW(r2, layout);
  GC_NEW(r3, layout);

  // Access objects
  GC_ASSIGN(r1, r2);           // r1 = r2
  GC_SET_REF(r3, ref1, r2);    // r1->ref1 = r2

  // Deallocate objects
  // No explicit deallocation is necessary
  // It is performed by the garbage collector

  // Remove roots from the root set
  GC_ROOT_BEGIN(layout);
  GC_ROOT_END(r3);
  GC_ROOT_END(r2);
  GC_ROOT_END(r1);

  // Termination of garbage collector and global variables
  GC_STATIC_END(layout);
  GC_EXIT;

  return 0;
}
```

Figure 4: The C–program example, to the left, utilises manual memory management. The other equivalent program is converted to the lowest layer of the garbage collector interface, i.e. without preemption. The focus in this layer is to handle all the references in a program.

Allocation

The primary and desired functionality of memory management is to allocate memory areas. Automatic memory management avoids burdening the programmer with the problems of deallocation. All the other macros, besides the allocation macro, are actually undesired, but necessary for the GC to handle the heap. Allocatable memory areas, objects, are described by their layouts. One of the primary functions of the layout is to locate the references in objects. All objects that are created from the same layout contain references at the locations that are described in the layout. In GCI, objects are allocated by `GC_NEW` and layouts are created by `GC_NEW_LAYOUT`.

The objects described by the GCI are as C–structures. However, the contents of the structure must be modified in a way that the garbage collector recognises. An object head is added in the structure that often contains information about the layout of the object, and other garbage collector specific information.

The layout is itself an object, with a reference to a *meta layout*. The apparent infinite number of layout referencing is actually terminated in a *meta meta layout* that describes not only its children but also itself (see Figure 5).

Garbage Collection Control

These macros handle the initialisation and destruction of the heap and the garbage collector. They have to be utilised only once during the lifetime of an application. The size of the heap is given as an argument to `GC_INIT`.

Reference Handling

The locations of the references are essential for the GC. Macros that are mainly related to the design of an object also contain information about the location of the references inside objects.

The macros in this category are divided into two parts: the declaration of objects and the creation of layouts. Ideally, there should only be the declaration of objects since the declaration contains all the locations of the references inside the object. The layout creation should be hidden from the programmer inside the object declaration. However, the macro–language is limited and forces explicit code for the layout creation. The example in Figure 2, the conversion of an ordinary C–structure to the GCI can be seen. The locations of the references inside the object are described during the creation of the layout (see the example in Figure 4).

Reference Declaration

The declaration of references must be covered by GCI because a reference has different outlooks in different GC algorithms. An important step in the declaration is to clear the location of the reference from old remnants. Before the reference is added to the reference graph, it must contain a valid reference. Old data could violate that criterion.

If the application is non–preemption, then references that are not registered may be utilised as long as the heap is kept intact, i.e. as long as the garbage collector does not perform any work. In systems with a single thread, the garbage collector performs work only during memory allocations. An unregistered reference is valid until a new object is allocated, in a non–preemptive application. Afterwards, it may be erroneous.

Reference Graph Modification

All the situations where references are added, removed, or changed, must be under the supervision of the GC. The reference graph is modified as roots are added or removed. Reference assignments also change the graph. References in the graph are also set when a new layout is created. Since these modifications of the reference graph may be implemented in various ways, it is covered by the GCI.

The Reference Graph Modifications are covered by the write barrier as preemption is added to the GCI. Hence the name Write Barrier in the Table 1.

Reference Graph Access

When values are accessed inside objects, the reference graph is also accessed. Since the implementation of objects may vary due to the GC algorithm, the implementation of the object access should also be hidden within the GCI.

The reference graph is accessed when references are compared, and when references are utilised in assignments. Access of values inside objects is the read barrier in the multithreaded applications. Table 1 utilises the word read barrier to be consistent with the preemptive layer.

3 Interface to the Code Generator

The utilisation of the complete layer of the garbage collector interface is intended for code generators (due to the high complexity). Only small programs like device drivers, or test applications should expose the GCI to the programmer. This layer covers preemption, i.e. mutual exclusion, and debugging. The thread and/or the debug layer may be switched off to suit the requirements of the application.

3.1 Mutual Exclusion

In preemptive multithreaded applications, the garbage collector can be interrupted anywhere during its execution. Two types of problematic interruptions may occur. First, the garbage collector algorithm may be interrupted during critical regions, e.g. memory copy. Second, interruption may occur while unregistered references are temporarily stored in processor registers, i.e. outside the reference graph.

The first type is handled by prohibition of rescheduling during critical regions. The garbage collector has to be supported by the surrounding system with procedures to enable and disable rescheduling. We will not discuss exactly where those critical regions are located in the GC code since they are algorithm specific. However, it is important to notice the necessary support of the surrounding system.

The second interruption is handled by the GCI and its overhead is hidden from the programmer. Several critical regions (all that we could think of) are covered by the interface. For example, the GC could be locked while *processor register references* are outside the reference graph. The garbage collector could be forced to *write through* the references to their memory locations, i.e. to the reference graph, before it is allowed to perform other duties. No other garbage collector work should be permitted (the GC is disabled) while references reside only in registers. Nevertheless, rescheduling is still allowed.

Another solution is to keep track of the reference registers, so that the garbage collector is able to identify all the references in an application at all times. Register references can be maintained by a *register map*, e.g. a specific register may be sacrificed to support the map. The compiler may be modified to produce the *map code*. More information about how a compiler may support garbage collection can be found in a paper written by [JR98]. The reference register map alternative has not been implemented yet for the GCI.

An example of a C–program that is adapted to the complete GCI is shown in Figure 2 (object declaration), Figure 6 (global reference), Figure 7 (function implementation), and Figure 8 (main program). The macros maintain disabling and enabling of the garbage collector due to preemption.

Critical regions where references reside in registers, occurs when roots are registered, new objects are created, and when a value inside an object is set

or fetched. Another situation where there may exist live references outside the reference graph is during procedure calls with reference parameters. The parameter references must be registered before the garbage collector is enabled. The garbage collector is disabled prior to procedure or function call. An example of the GCI function call macro is shown in Figure 8, where the function `saveref` is called with a reference parameter.

Nested expressions are not supported by the GCI macro to avoid compiler generated unregistered temporary reference variables. Both the destination and source references are arguments to the macros. Therefore, they cannot be nested. For example, the destination reference in an assignment is given as an argument to the `GC_ASSIGN`–macro together with the source reference. Temporary unregistered references shouldn't be generated by the compiler since the programmer is forced to declare all the temporary variables, and register them as roots.

Thread Control

The GC must be informed of the roots that every thread has. Ordinarily, the roots reside on the stack of the thread, and every thread has been allocated a stack. However, the implementation of how the roots of the threads are supervised, is hidden in the GCI.

Function Calls and Preemption

During an ordinary procedure or function call, the reference parameters are copied into locations determined by the compiler. The reference parameters must be registered as roots before the reference graph is up–to–date. Between the function call and the registration of the reference parameter, the GC must be disabled. The same situation arises if a function returns a reference. The GCI handles the return of a reference by explicitly emphasising the assignment of the return reference to a registered reference location. In Figure 7, the function `saveref` takes a reference argument and returns a reference to an object of the type `Object`. The complete procedure and function declaration is covered by the GCI, and extended with macros for parameter registration, function entrance, and function return. The macro that handles function return performs deregistration of the reference parameters.

Calls to functions that either return reference parameters or values have to be rewritten with the GCI macros: `GC_REF_FUNC_CALL`, or `GC_VAR_FUNC_CALL`, respectively. The macros also assigns the return value to the specified destination. If a procedure is called, `GC_PROC_CALL` is utilised.

The GC is disabled by the call macros and enabled, after the registration of reference parameters, with `GC_FUNC_ENTER`. As function returns a reference, the GC is disabled by the `GC_RETURN`–macro. The execution continues at the macro `GC_FUNC_LEAVE`, where the parameters are unregistered. Afterwards, the actual reference is returned and the GC is enabled by the function calling macro. A complete example of the GCI calling procedure is shown in Figure 7 and in Figure 8.

Write Barrier

The main purpose of the write barrier is to prevent mutations of the reference graph while the GC is working with it. Mutations happen when roots are added or removed, and during allocations. Another situation where the write barrier is necessary is during reference assignment. For instance, a reference that has been traversed by the GC, mustn't refer to an untraversed object. It should be noted that the write barrier is only relevant in copying and incremental garbage collectors.

Read Barrier

When values are accessed inside objects, the reference graph is accessed. Since the implementation of objects may vary due to the GC algorithm, the implementation of the object access should also be hidden within the GCI. The reference graph is also accessed when references are compared, and when references are utilised in assignments. Assignments have arguments that cover both the right value (access) and the left value (mutation). The reason for that is to prevent nested macro utilisation. The programmer is explicitly forced to add temporary references to hold intermediate results. This intentional overhead is implemented in the GCI to avoid unregistered temporary variables. Another reason is to limit the periods where the GC is disabled. Nested utilisation of macros that disable the GC could become very deep in automatically generated code, thus locking the GC for a long time. This contradicts the requirement of predictability in real–time programming. Nested locking also generates an overhead to determine when unlocking should occur. The number of unlocks must correspond to the number of locks before the GC can be enabled.

3.2 Debug Layer

The intricate and complex nature of garbage collection requires support for code analysis in the garbage collection interface, independent of the GC algorithm. This is implemented in the debug layer of the GCI. The debug checks may be switched off to remove the debug code overhead in, for example, a release version. The checks include, for instance, correct deallocation of roots, correct state of the garbage collector, and assertions that the references and values are well formed. The debug layer can easily be extended with more advanced analysis and assertions. If an error occurs, the type of error, the state of the garbage collector, as well as its location, are reported. The checks and debug reports effectively support implementation of new GC algorithms.

The state of the garbage collector is supervised by code in the debug layer of the GCI. They are utilised to determine if the GCI is utilised in the right context, and in the right order. For instance, the garbage collector cannot be utilised before it has been initialised. There are checks that ensure that the deregistration of roots are performed in the reverse order of their registration. Another complex situation where the order is significant is during procedure or function calls. The state of the garbage collector is utilised to determine if procedure or function calls are performed correctly.

In order to supervise the references, a separate debug graph of references is maintained by the debug layer. All reference declarations are extended with

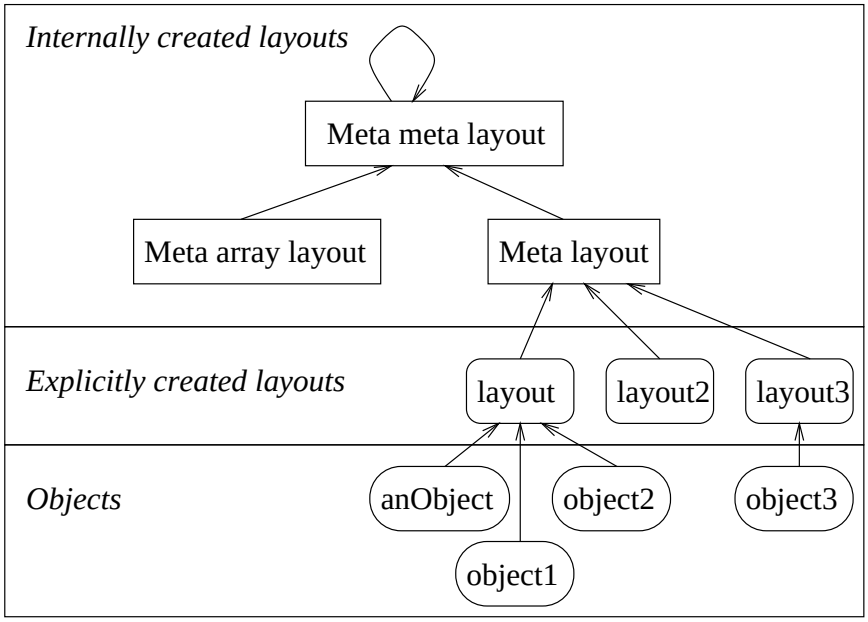

Figure 5: The layout structure shows how the objects and layouts in a program are connected. Layouts are also treated as objects.

extra references to support the debug graph. The extra references enable checks to ensure that references are live, and well formed. A well–formed reference refers to something that resembles an object.

If more memory is allocated than the size of the heap, it is important to inform the program user. However, that check is omitted from the debug layer because it is related to the garbage collector algorithm and not to the GCI.

3.3 Generation of Reference Locations

The locations of references in objects are important to describe. The description is stored in the layout of the object. The new layouts are connected to an internal *meta* layout structure that describes the reference locations inside layout objects. Figure 5 shows a more detailed description of the meta layout structure. An interesting detail is the description of the meta meta layout that describes its children as well as itself. More details about the meta layout description can be found in [Ive02].

The description of the location of references in an object must be independent of alignment[2] to ensure that it is possible to utilise the garbage collector on different platforms.

Figure 5 contains four objects. Three of the objects are instances of the layout named `layout` and the fourth object is an instance of `layout3`. All the layouts created in the program are described by their `meta layout`, which is in its turn described by the *meta meta layout*. The meta meta layout describes

[2]The internal organisation of C–structures does not have an unambiguously defined location of the structure entries. Furthermore, the size of pointers may vary on different platforms.

```
struct Object* saved;
```

```
GC_GLOBALS;
GC_STATIC_REF(Object, saved);
```

Figure 6: The global variable `saved` is declared as a reference to an object of type `Object`. Debugging is supported by the `GC_GLOBALS`–macro.

```
Object* saveref(int i, Object* r) {
  if ( !saved ) {
    r->o1 = saved;
    r->o3 = saved;
  }
  saved = r;
  {
    Object* tmp;
    tmp = saved;
    while ( !tmp ) {
      int val;

      val = tmp->i1;
      tmp = tmp->o1;
    }
  }
  return r;
}
```

```
// Declare a function named 'saveref' that returns a reference
// The function has two parameters: an integer and a reference
GC_REF_FUNC_BEGIN(Object, saveref, int i, GC_PARAM(Object, r))
  // Declare and register the parameter
  GC_PARAM_BEGIN(Object, r);
  // Mark that the function's code follow
  GC_FUNC_ENTER;

  if ( !GC_IS_NULL(saved) ) { // Check if 'saved' is not set
    GC_SET_REF(r, o1, saved); // Set entry o1 in r to saved
    GC_SET_REF(r, o3, saved); // Set entry o3 in r to saved
  }
  {
    GC_BEGIN_ROOT(Object, tmp);
    GC_ASSIGN(tmp, saved);       // Set tmp to saved.
    while ( !GC_IS_NULL(tmp) ) { // Check if tmp is not set
      int val;

      GC_GET(val, tmp, i1);
      GC_GET_REF(tmp, tmp, o1);
    }
    GC_END_ROOT(tmp);
  }
  GC_RETURN_REF(r);
  // Code here is never accessed
  GC_FUNC_LEAVE;
  GC_PARAM_END(r);
GC_REF_FUNC_END(Object, saveref);
```

Figure 7: A function implementation with the GCI.

itself and its children. The *meta array layout* is intended to describe arrays, however that is still ongoing work.

4 Garbage Collector Interface Specification

The Garbage Collector Interface, GCI, enables a common interface to garbage collector algorithms for multithreaded applications on single processor systems. The purpose is to support code generation with a unified way to express automatic memory management, and to incorporate garbage collection without deep knowledge of garbage collector and their algorithms. GC implementations can easily be linked together with software adapted to the GCI. The list below describes the utilisation of all the details of the GCI. The interface consists of C–macros.

Object Allocation

The allocation of an object is performed by the following macro.

`GC_NEW(var, layout)` Create an object from the description named `layout` and set the reference variable `var` to refer it.

Object Layout Declaration

The following macros are utilised to describe the structure of an object; a new object type is declared.

```
int main() {
  Object* ref1;
  Object* ref2;

  ref1 = (Object*) malloc(sizeof(Object));
  {
    int i;

    for (i=0; i<1000000; i++) {
      Object* r = (Object*) malloc(sizeof(Object));
      r->i1 = i;
      ref->o1 = r;
      if ( i%100000 == 0 ) {
        ref2 = saveref(i, ref1);
      }
    }
  }
  return 0;
}
```

```
int main() {
  GC_INIT(10000);          // Initialise GC and heap
  GC_THREAD_INIT;          // Initialise this thread
  GC_STATIC_BEGIN(saved); // Register the global ref as root

  // Declare and register references
  GC_ROOT_BEGIN(Object, ref1);
  GC_ROOT_BEGIN(Object, ref2);
  GC_ROOT_BEGIN(Layout, layout);

  // Create the description (layout) of Object
  GC_NEW_LAYOUT(layout, Object, 3,
    GC_LAYOUT_FIELD(Object, o1),
    GC_LAYOUT_FIELD(Object, o2),
    GC_LAYOUT_FIELD(Object, o3));

  GC_NEW(ref, layout);     // Create new 'layout'-object

  {
    int i;

    for (i=0; i<1000000; i++) {
      GC_ROOT_BEGIN(Object, r);
      GC_NEW(r, layout);  // Allocate new 'layout'-object
      GC_SET(r, i1, i);
      GC_SET_REF(ref, o1, r);
      if ( i%100000 == 0 ) {
        GC_REF_FUNC_CALL(ref2, saveref, i, GC_PASS(ref1));
      }
      GC_ROOT_END(r);      // Deregister 'r' as root
    }
  }
  GC_ROOT_END(layout);     // Deregister references as roots
  GC_ROOT_END(ref2);
  GC_ROOT_END(ref1);

  GC_STATIC_END(saved);    // Deregister static reference

  GC_EXIT();               // Deallocate heap and disengage
                           // the garbage collector
  return 0;
}
```

Figure 8: A complete program before and after the adaptation to the garbage collector interface

	GC_ROOT_BEGIN	GC_ROOT_END	GC_STATIC_REF	GC_STATIC_BEGIN	GC_STATIC_END	GC_SET	GC_GET	GC_SET_REF	GC_GET_REF	GC_SET_NULL	GC_PARAM	GC_PARAM_BEGIN	GC_PARAM_END	GC_NEW	GC_ASSIGN	GC_ASSIGN_NULL	GC_IS_NULL	GC_EQUAL	GC_INIT	GC_EXIT	GC_STRUCT_BEGIN	GC_STRUCT_END	GC_STRUCT_REF	GC_STRUCT_VAR	GC_NEW_LAYOUT	GC_LAYOUT_FIELD	GC_LAYOUT_ARRAY
Allocation														X												X	
GC control																			X	X							
Reference																					X	X	X	X	X	X	X
Declaration	X		X								X												X				
Write barrier	X	X		X	X			X		X		X	X	X	X	X									X		
Read barrier						X	X	X	X	X					X		X	X							X		
Preemption	X	X		X	X	X	X	X	X	X				X	X	X	X	X	X						X		
Debug	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X				X				

Table 1: The table shows the contents of the lowest layer of the GCI in connection to the functionality categories.

	GC_GLOBALS	GC_PROC_CALL	GC_REF_FUNC_CALL	GC_VAR_FUNC_CALL	GC_PASS	GC_PROC_BEGIN	GC_REF_FUNC_BEGIN	GC_VAR_FUNC_BEGIN	GC_FUNC_ENTER	GC_RETURN	GC_RETURN_REF	GC_RETURN_VAR	GC_FUNC_LEAVE	GC_PROC_END	GC_REF_FUNC_END	GC_VAR_FUNC_END	GC_THREAD_INIT	GC_THREAD_EXIT
Allocation																		
GC control																		
Reference																		
Declaration																		
Write barrier											X							
Read barrier											X							
Preemption		X	X	X			X		X						X		X	X
Debug	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Table 2: The table shows the contents of the GCI added due to the thread and debug layer of the GCI. The contents of the GCI is shown in connection to the functionality categories.

`GC_STRUCT_BEGIN(name)` Mark the beginning of an object declaration. The name of the object type is `name`.

`GC_STRUCT_END(name)` Mark the end of an object declaration. The described object is named `name`.

`GC_STRUCT_REF(type, name)` Declare a reference entry in the object. The type of the reference is `type`. The name of the reference is `name`.

`GC_STRUCT_VAR(type, name)` Declare a non–reference entry in the object. The type of the reference is `type`. The name of the reference is `name`.

The layout describes objects. It contains information common to all objects that are created from the layout. The information may be the object size and where the references are located inside the object.

`GC_NEW_LAYOUT(layout, object, #desc, desc1, ...)` Create a layout named `layout` for the reference type `object`. The number of references inside the object is equal to `#desc`. The locations of the references inside the object are described by the descriptors, i.e. `desc1, ...`. There are two types of descriptors:

`GC_LAYOUT_FIELD(object, field)` The reference named `field` is in the type `object`. *Note:* `object` is the same type as stated in the enclosing `GC_NEW_LAYOUT`–block.

`GC_LAYOUT_ARRAY(field)` The elements *after* the field named `field` are located in an array. The array may be of different sizes and the number of entries in the array is stored in the object and not inside the layout.

Initialisation and Termination

These macros concern the initialisation and destruction of the garbage collector, and the threads that are under the supervision of the GC. They are utilised once for each application, or thread.

`GC_GLOBALS` Declaration of internal garbage collection global variables is handled by this macro. It must always be declared once in a program adapted to GCI, whether or not global references are utilised or not.

`GC_INIT(heapsize)` Initialise the garbage collector (the internal meta structure is created) and set the size of the heap to `heapsize`. This macro is utilised once and before the first utilisation of the heap.

`GC_EXIT` Remove the heap and shut down the garbage collector.

`GC_THREAD_INIT` Inform the garbage collector of this thread. The roots inside the thread are made known to the garbage collector.

`GC_THREAD_EXIT` Inform the garbage collector that the thread no longer has to be under the supervision of the garbage collector.

Reference Declaration

Declaration of static (global) and local reference variables are covered with these macros. Registration and deregistration is also handled.

`GC_STATIC_REF(type, name)` Declare a static reference variable named `name` with the type `type`.

`GC_STATIC_BEGIN(var)` Make the static reference variable named `var` known to the garbage collector. The reference will be registered as a root.

`GC_STATIC_END(var)` Remove the registered static reference variable named `var` from the supervision of the garbage collector. The variables must be removed in the reverse order of registration.

`GC_ROOT_BEGIN(type, var)` Declare a reference variable named `var` with the type `type` and make the reference variable known to the garbage collector. The reference will be registered as a root.

`GC_ROOT_END(var)` Remove the registered reference variable named `var` from the supervision of the garbage collector. The variables are removed by the GCI in the reverse order of registration.

Function Utilisation

These macros concern procedure and function declaration and the registration that concern references. The garbage collector must be disabled during a procedure or function call that contains reference parameters, and while the function returns a reference. Reference parameters must be registered as roots before the function is entered.

`GC_PROC_CALL(proc_name, ...)` Call a procedure named `proc_name`. Arguments are stated after the `proc_name` (at the ellipsis).

`GC_REF_FUNC_CALL(result, func_name, ...)` Call a function named `func_name` that returns a reference. The reference is stored in the reference variable `result`. Arguments are stated after the `func_name` (at the ellipsis).

`GC_VAR_FUNC_CALL(result, func_name, ...)` Call a function named `func_name` that returns a non–reference value. The value is stored in the variable `result`. Arguments are stated after the `func_name` (at the ellipsis).

`GC_PASS(name)` Inform the GC that the argument named `name` in the procedure or function call, is a reference.

`GC_PROC_BEGIN(proc, ...)` Declare a procedure named `proc`. Parameter declarations are written after the name of the procedure (ellipsis).

`GC_REF_FUNC_BEGIN(type, func, ...)` Declare a function named `func` and that returns a reference of type `type`. Parameter declarations are written after the name of the function (at the ellipsis).

`GC_VAR_FUNC_BEGIN(type, func, ...)` Declare a function named `func` and that returns a non–reference value of type `type`. Parameter declarations are written after the name of the function (at the ellipsis).

`GC_PARAM(type, name)` Declare a parameter named `name` that is a reference to an object of type `type`.

`GC_PARAM_BEGIN(param)` Inform the garbage collector of the reference parameter `param`.

`GC_FUNC_ENTER` Inform the garbage collector of the start of the procedure or function code. All the reference parameters must be registered before this point. *Note:* The garbage collector is enabled here in a multithreaded application.

`GC_RETURN` Return from the procedure. Execution will continue at `GC_FUNC_LEAVE`.

`GC_RETURN_REF(result)` Return the reference `result` from the function. Execution will continue at `GC_FUNC_LEAVE`.

`GC_RETURN_VAR(result)` Return the non–reference value `result` from the function. Execution will continue at `GC_FUNC_LEAVE`.

`GC_FUNC_LEAVE` Mark the end of the function code. Reference parameters should be deregistered after this mark.

`GC_PARAM_END(param)` Deregister the reference parameter `param`. They must be deregistered in the reverse order of registration.

`GC_PROC_END(proc)` Mark the end of a procedure declaration named `proc`.

`GC_REF_FUNC_END(type, func)` Mark the end of a function declaration named `func` that returns a reference of the type `type`.

`GC_VAR_FUNC_END(type, func)` Mark the end of a function declaration named `func` that returns a non–reference value of the type `type`.

Reference Access

The utilisation of references, or reference access, is covered by these macros.

`GC_ASSIGN(destination, source)` Make the reference named `destination` refer to the same object that the reference named `source` refers to.

`GC_ASSIGN_NULL(ref)` Set the reference named `ref` to nothing.

`GC_IS_NULL(ref)` Check if the reference named `ref` is referring to nothing. Return zero if the reference named `ref` is referring to an object, otherwise return a non–zero value.

`GC_EQUAL(res, ref1, ref2)` Check if the references named `ref1` and `ref1` refer to the same object. Store the result in the variable `res`. If the equivalence is not valid, the resulting variable is set to zero. Otherwise, a non–zero value is stored.

Field Access

These macros cover the access to fields inside objects. The macros are divided into value macros and reference macros, because they are treated differently.

`GC_GET(var, object, field)` Set the variable `var` to the value of the field named `field` inside the object named `object`. *Note:* This is similar to the C–code, `var = object->field`.

`GC_SET(object, field, value)` Set the field named `field` inside the object named `object` to the value named `value`. *Note:* This is similar to the C–code, `object->field = var`.

`GC_GET_REF(var, object, field)` Make the reference variable named `var` refer to the same object as the reference named `field` that is inside the object named `object`. *Note:* This is similar to the C–code, `var = object->field`.

`GC_SET_REF(object, field, ref)` Make the reference named `field` inside the object named `object` refer to the same object as the reference named `ref` is referring to. *Note:* This is similar to the C–code, `object->field = ref`

`GC_SET_NULL(object, field)` Set the reference named `field` inside the object named `object` to nothing.

5 Results

The GCI stems from many earlier GC implementations and different requirements, e.g. to verify real–time GC implementations. For the GCI specifically, almost the complete interface has been implemented as an incremental mark–compact GC, in a C–code generator for the Caltrop system (model based integration of embedded software), [CEJW02], [Wer02]. Some macros are not completely implemented in the code generator, but the essential sections have successfully been implemented, i.e. the generated C–code is solely utilising the

GCI. Preemption may be executed anywhere outside the critical regions of the GC.

The GCI serves as an important role in the generation of code in the Java–to–C code converter written by Anders Nilsson, [Nil]. The Java–to–C converter is aimed towards generation of C–code for various tiny embedded systems from portable Java source code. We are in the process of utilising the GCI to accomplish the necessary portability in the Java–to–C converter. The ability to change, modify, or implement GC algorithms for tiny embedded systems is crucial because of the restricted environment.

The lower layer of the GCI is implemented as the essential memory manager in a Java Virtual Machine research project called Infinitesimal Virtual Machine, *IVM*, [Ive02]. Java programs constitute of *Java bytecodes* that are interpreted by the IVM. In the current IVM implementation preemption points are inserted after the execution of every (or a number) of bytecodes. Currently, preemption cannot occur inside *native* C–code inside the IVM. Ongoing work will implement the preemption layer of the GCI in the IVM.

Three different GCI adapted (lower layer) garbage collector implementations may be interchanged within the IVM. A command to the installation program of the IVM selects the desired garbage collector. The implemented garbage collectors are an incremental mark–compact, an incremental batch–copy mark–sweep, and a combination of the two previously mentioned garbage collectors.

A simple malloc–free implementation has also been implemented in the GCI.

6 Conclusions

There is a common misconception, both in academia and in the software industry, that the presence of GC hinders software to exhibit real–time properties. From earlier work we knew that that is not true, which is fortunate since the use of GC is necessary for the use of safe languages. In particular, it is highly desirable to enable the use of a safe language, such as Java, in the embedded world where many systems are called critical.

However, opposed to desktop or server computers, embedded computers show a great variation in type of CPU, amount of memory, timing requirement, IO interfaces, and the like. Therefore, despite the fact that most CPUs are embedded, we cannot expect standard Java (or RT–Java) platforms to be available. Instead, we have to find techniques to provide Java on top of other languages. Instead of Java, one could of course do the same thing for C#, but Java is the natural platform independent safe programming language. As the low–level language, C is the only reasonable choice since it is the only language available for all (?) types of hardware, and its properties turned out to suit our purposes well. Thus, for the success of proper object orientation in the embedded world, it is crucial to find a generic and efficient GCI for different types of execution scheduling.

To our knowledge, there has not existed such a GCI until now. Clearly, the GCI is not intended for manual programming in the target C–language, even if that can be done when writing small device drivers. Instead, the GCI should primarily be used together with generated code (from Java or from modelling tools), and in same cases by implementers of virtual machines and/or debugging tools.

After extensive investigations and prototyping, we claim that the proposed GCI provides the needed functionality and without harming run–time efficiency of timing properties significantly. In other words, we found it possible to define a generic GC interface encapsulating the technical details of different GC algorithms and run–time properties.

References

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[CEJW02] Chris Chang, Johan Eker, Jörn W. Janneck, and Lars Wernli. *Caltrop developer's handbook*. Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, 2002.

[gcc00] *GNU Compiler Collection*, 2000. *http://gcc.gnu.org*.

[Hen98] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, July 1998.

[Ive02] Anders Ive. Implementation of an embedded real–time java virtual machine prototype. Licentiate thesis, Department of Computer Science, Lund Institute of Technology, 2002.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[JR98] Simon Peyton Jones and Norman Ramsey. Machine–independent support for garbage collection, debugging, exception handling, and concurrency. *http://www.eecs.harvard.edu/ nr/pubs/c–rti-abstract.html*, August 1998.

[Nil] Anders Nilsson. Java–to–C conversion for tiny embedded systems. *anders.nilsson@cs.lth.se*.

[Wer02] Lars Wernli. Design and implementation of a code generator for the CAL actor language. Master's thesis, University of California at Berkeley, April 2002.