Exact Roots for a Real-Time Garbage Collector

Rasmus Pedersen
Department of Informatics
CBS, Denmark
rup.inf@cbs.dk

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

ABSTRACT

Garbage collection is traditionally not used in real-time systems due to the unpredictable temporal behavior of current implementations of a garbage collector. However, without garbage collection the programming model is very different from standard Java. It is the opinion of the authors that garbage collection algorithms can be adapted to meet even the requirements for hard real-time systems.

One important property of a real-time garbage collector is to identify only the *real* roots on the root scan. Misinterpreting primitive values as false root pointers can result in an unpredictable worst case memory consumption. In this paper we propose a method to add information on the stack layout to the runtime data structure in order to find the roots exactly. Furthermore, interpreting this information during the collection process is implemented to be worst-case execution time analyzable.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.2 [**Programming Languages**]: Java; D.3.4 [**Programming Languages**]: Processors—Memory management (garbage collection)

General Terms

Algorithms, Languages

Keywords

Garbage Collection, Real-Time, Java, Root Set

1. INTRODUCTION

Garbage Collection (GC) is an essential part of the Java runtime system. GC enables automatic dynamic memory management, which is essential to build large applications. Automatic memory management frees the programmer from complex and error prone explicit memory management (`malloc` and `free`).

Dynamic memory allocation in hard real-time systems is attractive as the programmer can use the same style and patterns as with non-real-time programs that use traditional garbage collectors. In real-time systems and especially in embedded real-time systems there are some additional constraints on the garbage collector. It needs to be real-time analyzable with guarantees that it can be scheduled with other real-time threads. This demand can motivate the need for time-predictable garbage collection.

A garbage collector in Java collects objects on the heap that are not reachable from the application anymore. The garbage collector searches the global variables (the class variables) and the Java stack frames for objects references. These references are called the *root set*. Starting from this root set the complete graph of all live objects is traversed. The objects that are not found during this traversal are not referenced anymore and thus are garbage.

A prerequisite for a time-predictable root set scan is that this initial set of object references is precise and does not contain any false references. It is such that the primitive values in Java like `int` can look like a reference to an object without being one. This can fool the garbage collector to retain this object as a live object and in turn break the prerequisites for the scheduling of a concurrent real-time garbage collector. Accordingly, we implement and analyze an exact root set scanner, which is configurable in terms of its memory overhead depending on the application programmers scheduling requirements.

The paper is structured as follows: Section 2 provides the overview of real-time garbage collection (RTGC) and related work. In Section 3 we describe the architecture of the open source Java processor used for the implementation of the exact root set scanner and how some architectural choices influence the root set. Section 4 details the design of the exact root set scanner and provides the experimental results in Section 5. We continue with the discussion of the root set scanner in Section 6 and concluded the paper in Section 7.

2. REAL-TIME GARBAGE COLLECTION

Garbage collection is considered unsuitable for real-time systems due to the unpredictable blocking times introduced by the GC work. As one solution to use Java for real-time systems the Real-Time Specification for Java (RTSJ) [6] introduces new thread types with program managed, scoped memory for dynamic memory requirements. The RTSJ programming model differs largely from standard Java and is difficult to use [20, 26]. It is the opinion of the authors that a garbage collector with real-time properties is mandatory to build future, probably more complex, real-time applications in Java.

To distinguish between other garbage collectors and a collector for (hard) real-time systems we cite from [32] the definition of real-time collector:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JTRES '06 October 11-13, 2006 Paris, France
Copyright 2006 ACM 1-59593-544-4/06/10 ...$5.00.

A real-time garbage collector provides time predictable automatic memory management for tasks with bounded memory allocation rate with minimal temporal interference to tasks that use only static memory.

The exact root scanner presented in this paper is one essential part of a real-time enabled garbage collector.

2.1 First Collectors

Garbage collection was first introduced for list processing systems (LISP) in the 1960's [19]. The first collectors were *stop-the-world* collectors that are called when a request for a new element can not be fulfilled. The collector, starting from pointers known as the root set, scans the whole graph of reachable objects and marks these objects live. In a second phase the collector *sweeps* the heap and adds unmarked objects to the free list. On the marked objects, which are live, the mark is reset in preparation for the next cycle.

However, this simple sweep results in a fragmented heap which is an issue for objects of different sizes. An extension, called *mark-compact*, moves the objects to compact the heap instead of the sweep [8]. During this compaction all references to the moved objects are updated to point to the new location.

Both collectors need a stack during the marking phase that can grow in the worst-case up to the number of live objects. Cheney [7] presents an elegant way how this mark stack can be avoided. His GC is called *copying-collector* and divides the heap into two spaces: the *to-space* and the *from-space*. Objects are moved from one space to the other as part of the scan of the object graph.

2.2 Incremental Collection

All the former described collectors are still stop-the-world collectors. The pause time of up to seconds in large interactive LISP applications triggered the research on incremental collectors that distribute collection work more evenly [35, 10, 4]. These collectors were sometimes called *real-time* although they do not fulfill hard real-time properties that we need today.

Baker [4] extends Cheneys [7] copying collector for incremental GC. However, it uses an expensive read barrier that moves the object to the to-space as part of the mutator work. Baker proposes the *Treadmill* [5] to avoid copying. However, this collector works only with objects of equal size and still needs an expensive read barrier.

A good overview of GC techniques can be found in [16] and in the GC survey by Wilson [36].

2.3 Real-Time

Work on real-time garbage collectors started again through consideration of Java for real-time systems [17].

In [25] two collectors based on [10] and [5] are implemented on a multithreaded microcontroller. Higuera suggests in [14] the use of hardware features from picoJava [24] to speed up RTSJ memory region protection and garbage collection. A hardware assisted real-time collector is proposed by Schmidt and Nielsen [29].

In [28] the authors suggest a time-triggered garbage collector and provide an upper bound for the GC cycle time. An extension on this topic can be found in [32].

The collector used for the root scanning presented in this paper is based on the work by Steele [35], Dijkstra [10] and Baker [4]. However, the copying collector is changed to perform the copy of an object concurrent by the collector and not as part of the mutator work. Therefore we name it *concurrent-copy* collector.

It is the opinion of the authors that garbage collection is an option for real-time systems. In [32] we showed that a periodic scheduled garbage collector can keep up with the demands of real-time threads. This is also the case for [21], which focus on portable solutions using JIT. The exact root scan for JOP, with the low and bound blocking time, as presented in this paper is another step towards enabling Java for real-time systems.

2.4 Exact Roots

Next, we will discuss exact root set scanning. There are a number of different approaches to identify the potential references in a stack. We will discuss a *split stack*, a runtime *type stack*, and a *stack map*. The first two are online approaches that keep track of the references in the embedded JVM [15]. The stack map method is an offline approach that use information generated from the class file during link time [1]. Others [11] have looked into the compression of this compile-time information with good results. In addition, there is also the method presented in [34], which requires the use of *synchronization points*. At such a synchronization point it is required that live references are stored on the heap. For a hardware implementation of the JVM (a Java processor) that contains the stack on-chip the additional type stack (which is just an additional 33^{rd} bit) is a valuable option [13].

Garbage collection in real-time Java systems is useful because it enables the use of dynamic memory allocation while the system still adheres to hard real-time requirements. In order to allow allocation of new objects the GC must be able to detect dead objects on the heap. In order to detect which objects are not reachable from the root set, the GC has to know which objects are live as the rest must then be dead and subject to garbage collection.

A garbage collector that scans objects for references to build a graph of live objects needs an initial set of references. This set is called the root set and is identified by inspecting the static variables and the stack frames of the program. These stack frames belong to different threads and are not shared among threads. A Java stack frame corresponds to one method in a class. It does not matter if the method is static or an instance method except that the instance method has the reference to the instance as the first argument (which is mapped to the first local variable during method invocation).

A root scan needs to inspect each frame for potential references to objects (which are located in the heap memory area) to get the initial set of references needed to make a live graph of the objects. The objects that are not reachable starting from the root set of references are dead in the sense that they can never be used again. They can safely be collected by the garbage collector. Each GC cycle is thus initiated with the identification of the root set of objects on the heap. The root set contains all static references (class variables) and the references on the thread's stacks. From this root set the rest of the reachable heap objects are identified.

3. A REAL-TIME JAVA PROCESSOR

In this section we introduce the environment where the garbage collector with the exact root scan is implemented. Furthermore, some features of the Java processor, such as only interrupts at bytecode boundary, simplify the exact root finding process.

The Java Optimized Processor (JOP) [31] is an implementation of the Java virtual machine (JVM) in hardware – a Java processor. JOP is a stack computer with its own instruction set, called microcode. Java bytecodes are translated into microcode instructions or sequences of microcode. The difference between the JVM and JOP is best described as the following: The JVM is a CISC stack architecture, whereas JOP is a RISC stack architecture.

JOP is implemented in a field programmable gate array (FPGA). One design goal for JOP is its applicability to worst-case execution time (WCET) analysis. This design principle is consistent throughout the JOP processor. As the processor is implemented in an FPGA and all sources are available, it is also possible to add specialized hardware units (e.g. a typed stack as suggested in Section 2.4. The thread scheduler on JOP [31] is preemptive. On a control switch a complete stack belonging to the active thread is saved. Therefore is can be easily accessed by the root scanner. Each frame on the stack contains the saved program counter (PC) that points to the next instruction that will get executed once control returns from the invoked method.

3.1 The Processor Pipeline

JOP is a fully pipelined architecture with single cycle execution of microcode instructions and a novel approach to mapping Java bytecode to these instructions.

Three stages form the JOP core pipeline, executing microcode instructions. An additional stage in the front of the core pipeline fetches Java bytecodes – the instructions of the JVM – and translates these bytecodes into addresses in microcode. Bytecode branches are also decoded and executed in this stage. The second pipeline stage fetches JOP instructions from the internal microcode memory and executes microcode branches. Besides the usual decode function, the third pipeline stage also generates addresses for the stack RAM. As every stack machine instruction has either *pop* or *push* characteristics, it is possible to generate fill or spill addresses for the *following* instruction at this stage. The last pipeline stage performs ALU operations, load, store and stack spill or fill. At the execution stage, operations are performed with the two topmost elements of the stack.

A stack machine with two explicit registers for the two topmost stack elements and automatic fill/spill needs neither an extra write-back stage nor any data forwarding. Details of this two-level stack architecture are described in [30]. The short pipeline results in short branch delays. Therefore, a hard to analyze, with respect to Worst Case Execution Time (WCET), branch prediction logic can be avoided.

3.2 Interrupt Logic

Interrupts are considered hard to handle in a pipelined processor, meaning implementation tends to be complex (and therefore resource consuming). In JOP, the bytecode-microcode translation is used to avoid having to handle interrupts in the core pipeline.

Interrupts are implemented as special bytecodes. These bytecodes are inserted by the hardware in the Java instruction stream. When an interrupt is pending and the next fetched byte from the bytecode cache is an instruction, the associated special bytecode is used instead of the instruction from the bytecode cache. The result is that interrupts are accepted at bytecode boundaries. The worst-case preemption delay is the execution time of the *slowest* bytecode that is implemented in microcode. Bytecodes that are implemented in Java (see Section 3.3) can be interrupted.

The implementation of interrupts at the bytecode-microcode mapping stage keeps interrupts transparent in the core pipeline and avoids complex logic. Interrupt handlers can be implemented in the same way as standard bytecodes are implemented i.e. in microcode or Java. This special bytecode can result in a call of a JVM internal method in the context of the interrupted thread. This mechanism implicitly stores almost the complete context of the current active thread on the stack.

It has to be noted that the interrupt is only handled at bytecode boundaries. We do not need to consider interruption of the microcode which greatly simplifies the implementation of the root scanning. At bytecode boundaries no valid references in processor internal registers have to be considered. For a compiling JVM (a JIT compiler or a batch compiler [22]) also the registers of the processor have to be considered as potential root references or explicit synchronization with the collector has to be included [23].

3.3 Microcode

The following discussion concerns two different instruction sets: *bytecode* and *microcode*. Bytecodes are the instructions that make up a compiled Java program. These instructions are executed by a Java virtual machine. The JVM does not assume any particular implementation technology. Microcode is the native instruction set for JOP. Bytecodes are translated, during their execution, into JOP microcode. Both instruction sets are designed for an extended[1] stack machine.

3.3.1 Translation of Bytecodes to Microcode

To date, no hardware implementation of the JVM exists that is capable of executing *all* bytecodes in hardware alone. This is due to the following: some bytecodes, such as `new`, which creates and initializes a new object, are too complex to implement in hardware. These bytecodes have to be emulated by software.

To build a self-contained JVM without an underlying operating system, direct access to the memory and I/O devices is necessary. There are no bytecodes defined for low-level access. These low-level services are usually implemented in *native* functions, which mean that another language (C) is native to the processor. However, for a Java processor, bytecode is the *native* language.

One way to solve this problem is to implement simple bytecodes in hardware and to emulate the more complex and *native* functions in software with a different instruction set (sometimes called microcode). However, a processor with two different instruction sets results in a complex design.

In JOP, this problem is solved in a much simpler way. JOP has a single *native* instruction set, the so-called microcode. During execution, every Java bytecode is translated to either one, or a sequence of microcode instructions. This translation merely adds one pipeline stage to the core processor and results in no execution overheads. With this solution, we are free to define the JOP instruction set to map smoothly to the stack architecture of the JVM, and to find an instruction coding that can be implemented with minimal hardware.

Every bytecode is translated to an address in the microcode that implements the JVM. If there exists an equivalent microinstruction for the bytecode, it is executed in one cycle and the next bytecode is translated. For a more complex bytecode, JOP just continues to execute microcode in the subsequent cycles. The end of this sequence is coded in the microcode instruction. This translation needs an extra pipeline stage but has zero overheads for complex JVM instructions.

3.3.2 Compact Microcode

For the JVM to be implemented efficiently, the microcode has to *fit* to the Java bytecode. Since the JVM is a stack machine, the microcode is also stack-oriented. However, the JVM is not a pure stack machine. Method parameters and local variables are defined as *locals*. These locals can reside in a stack frame of the method and are accessed with an offset relative to the start of this *locals* area. Additional local variables (16) are available at the microcode level. These variables serve as scratch variables, like registers in a conventional CPU. However, these JVM local variables do not retain any object related information over bytecode boundaries. As explained in Section 3.2 those variables are not part of the root set.

[1] An extended stack machine contains instructions that make it possible to access elements deeper down in the stack.

Some bytecodes, such as ALU operations and the short form access to *locals*, are directly implemented by an equivalent microcode instruction (with a different encoding). Additional instructions are available to access internal registers, main memory and I/O devices. A relative conditional branch (zero/non zero of TOS) performs control flow decisions at the microcode level. For optimum use of the available memory resources, all instructions are 8 bits long. There are no variable-length instructions and every instruction, with the exception of `wait`, is executed in a single cycle.

3.3.3 Flexible Implementation of Bytecodes

As mentioned above, some Java bytecodes are very complex. One solution already described is to emulate them through a sequence of microcode instructions. However, some of the more complex bytecodes are very seldom used. To further reduce the resource implications for JOP, in this case local memory, bytecodes can even be implemented by *using* Java bytecodes. During the assembly of the JVM, all labels that represent an entry point for the bytecode implementation are used to generate the translation table. For all bytecodes for which no such label is found, i.e. there is no implementation in microcode, a *not-implemented* address is generated. The instruction sequence at this address invokes a static method from a system class (`com.jopdesign.sys.JVM`). This class contains 256 static methods, one for each possible bytecode, ordered by the bytecode value. The bytecode is used as the index in the method table of this system class. This feature also allows for the easy configuration of resource usage versus performance.

The bytecodes that create objects (e.g. *new*) are also implemented in Java for a simpler interaction with the GC[2]. As this bytecode could be interrupted by a thread switch it is explicitly synchronized with the GC thread.

4. EXACT ROOT SET SCAN

The runtime information of all classes is structured in a way that simplifies scanning of the static references. The class variables are divided into primitive type and reference type variables. All reference variables are found in one continues block in the memory. Therefore, scanning the static references is trivial. Identifying exactly the stack elements that contain a reference is more challenging.

First we need to discuss conservative stack scanning. It also operates with the goal of identifying the root set. A simple approach is to handle each stack value as a potential root. In this way it examines each value on the stack and tests if it contains a value that points to either a handle (if indirection is used) or directly to an object. This is named conservative root scanning and can keep objects artificially alive through misinterpreting a primitive element as a reference. A value on the stack that belongs to a primitive can accidently equal a reference value to an object that is not referenced by any real references and thus *should* be garbage collected.

Another disadvantage of the conservative root set scan is that it is not possible to move objects when no indirection (the read barrier, or handle) is used. When the heap is compacted all references to the moved object have to be updated. An update of a misinterpreted stack slot would be wrong.

The first option is to work with a split stack [15] such that one stack is used only for references (ie. the root set) and the other stack is used only for primitives. The main drawback of this option is the need for additional typed bytecode instructions for the type-less bytecodes (e.g. `dup, pop`) that manipulate the stack. Furthermore, the split stack solution complicates the frame handling on `invoke` and `return` resulting in high overheads for method invokation.

A second option is to save this type information during runtime using a second stack (that can be one bit width as we only need to distinguish references from primitives), which contains the type information. This *type stack* has to be manipulated simultaneously with the value stack. In this case the type-less bytecodes are not an issue anymore. However, implementing this additional stack in software introduces a lot of overhead.

The third method is to extract this information from the class file (and the bytecodes) offline. This is possible due the fact that each PC maps to a unique configuration of the operand stack and local variables [12]. It is done by simulating the program execution while keeping track of the local variables and the stack operands. Accordingly, this information makes it possible to mark which locals or stack operands are primitives and which are references to a heap object. A requirement for this to work is that the program counter (PC) is mapped to a record that contains this information. It should be noted that this stack map could also be constructed during run time but it would be likely to consume too many resources to be feasible.

4.1 Program Counter Mapping

On the host where the Java application is compiled for the target, we perform the program simulation that maps each PC to the type (primitive or reference) in the local variables and the stack. We build a table that maps each possible PC value inside a method to the information about the stack layout.

There are two kinds of variables in each stack frame that potentially can hold references: the local variables and the stack operands. Common for both is that the type changes during execution. For example, a local variable in a given slot is used for a given interval of the PC. This interval is not necessary the full length of the method code. Furthermore, Chap. 7.2 in the Java Virtual Machine specification encourages reuse of local variables [18].

However, it is not common that a reference type variable is reused in a primitive type variables place, but it can be the case. Therefore, in order to scan the root set, we need to know the type (reference or primitive) of each local variable for each value of the PC. The stack operands are the second source of references. The operands can also be references and need to be included into the root set. Even more than the local variables the operands changes very often for each value of the PC. So we need the same PC to operand mapping such that it is known which of the operands are references.

In summary, a Java program can be annotated with information that contains information regarding the type of the local variables and the stack operands for a given value of the PC. This information can be made available to the embedded JVM in a number of ways.

4.2 Implementation

The local and operand type information for each PC can be obtained using a JVM stack simulator as described before. We have created an open source stack simulator based on the verifier[3] from the *Byte Code Engineering Library* [9] that verifies the Java class according to the rules in *Pass 3*, Chap. 4.9.1 the JVM specification version 2 [18].

With this stack simulator we extract the type information for each local variable and each stack operand for each value of the PC. That is the main flow of the simulator. Then there are a few but important exceptions that must be dealt with. First, our simulator detects if the type (reference or primitive) of a local variable slot

[2]The Java implementation of *new* in `JVM.java` actually invokes a method from the GC class

[3]`...bcel.verifier.structurals.Pass3bVerifier`

depends on the control path leading to this PC. This control path property is the so-called *Gosling* property, which is also discussed in [1]. It depicts that the local variables and the stack operands must contain the same types regardless of how a given PC was reached. Or in other words: The variable type is not dependent on the control flow. This property simplifies the verification of Java class files and also the generation of the stack map.

However, one exception to this *Gosling* property is allowed: The code that is executed for `finally` is implemented by a form of *mini-method* [1] that gets called by the bytecode `jsr`. It is allowed that this code is reached from different paths with different stack type layouts. Therefore the type information is control flow dependent. It is rare and we did not encounter it in our experiments, which is expected as we do not use exceptions in the JVM. One solution to this issue is to inline [3] the bytecode so the control flow independency also holds for these *mini-methods*. Furthermore, Sun's Java compilers version 1.4.2 and later compile `finally` blocks without subroutines.

4.3 Stack Map Packing

A possible problem is that the type information for the locals and the operands for each PC can be verbose to the extent that it prohibits an effective implementation. The direct (but verbose) way to pack this information is to create a small record with the same number of bits as the maximum number of locals and stack operands. These small records are then appended for each value of the PC for each method. We would also need to pad the operand(s) position(s) in the bytecode with the same bit pattern as the PC of the instruction (or any bit pattern as these positions are not accessed).

The second way to pack this reference marking information is to realize that the bit patterns formed by appending the marked local variables and the operands generates very limited number of different patterns for the PCs of each method. We can construct the PC to operand mapping using the same technique as with the Java constant pool. The PCs of the method are now an index into this small pool of unique patterns.

However, we need additional information in the target to make the stack maps work. One piece of needed information is the number of instructions for each method. The second and third pieces of information are the maximum number of local variables and stack operands. We save this information in a 32 bit word just before the method code address and at run time we can access this information word. It also marks if the method has any type information in the memory just below this information word. This word is not counted as overhead as the method information structure of the JOP JVM system can hold the necessary information; it is implementation specific.

To explain this PC mapping we provide an example. The source code for this example is from the `char charAt(int index)` method in `String.java` that compiles to the bytecodes shown in Figure 1. Next, we will look at the stack properties of this code and analyze how each of the slots for the local variables and each of the slots for the operands looks for each value of the PC. The *this* reference is in the first local slot throughout the code as depicted in the associated *LocalVariableTable* in the Figure 1.

Table 1 shows the result of the analysis for `java.lang.String.charAt(I)C` in Figure 1. The column PC shows the address of the instruction (in column Instr.). The `charAt` method has 2 arguments and maximum operand stack size is 2; that is what the next 4 columns show. The last column, p, enumerates the unique bit patterns. In this example there are 3 unique patterns each of length 4 bits giving 12 bits for the pattern pool of this method. Now each PC has to be mapped to the corresponding pattern and with 3 unique patterns the requirement for the index width becomes 2 bits. Since the method length is 7 we need 14 bits for the indices. In total we have used 26 bits. Comparing to directly encoding the bits for the 2 locals and the 2 operands would have resulted in 28 bits. In this example, the encoding saved 2 bits. For longer methods and for larger numbers of arguments, locals and operands the results will be more significant, which will be demonstrated empirically later.

In Figure 2 it is shown how the code of Table 1 is packed. The figure outlines why this can lead to reduction in the packed information because PC {1-4} point to the same pattern. It should be noted that PC {2,3} are just padding up the operands of PC 1.

An optimization for the memory consumption would be to store the stack map only for PC values that actually point to the start of an instruction. This would avoid the padding as seen in the former example. However, the calculation of the correct index into the table would be very time consuming as the code from the method start up to the current PC value needs to be analyzed.

```
Source code from String.java:
public char charAt(int index) {
  return value[index];
}

Output from javap -c -verbose:
public char charAt(int);
  Code:
   Stack=2, Locals=2, Args_size=2
   0:   aload_0
   1:   getfield        #3; //Field value:[C
   4:   iload_1
   5:   caload
   6:   ireturn
  LocalVariableTable:
   Start Length Slot Name  Signature
   0     7      0    this  Ljava/lang/String;
   0     7      1    index       I
```

Figure 1: Example bytecode from String.java

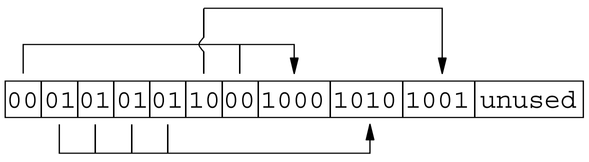

Figure 2: Packing the GC info

Table 1: Indexed program counter

PC	Instr.	l[0]	l[1]	o[0]	o[1]	p
0	aload_0	1	0	0	0	1
1	getfield	1	0	1	0	2
2	-	1	0	1	0	2
3	-	1	0	1	0	2
4	iload_1	1	0	1	0	2
5	caload	1	0	0	1	3
6	ireturn	1	0	0	0	1

Table 2: Stack Map Memory Usage (words)

	All			Only Ref.		
	Raw	Indx.	R.Indx.	Raw	Indx.	R.Indx.
Hello	3,033	1,158	1,132	1,916	972	960
DoAll	4,503	1,894	1,859	3,194	1,621	1,601
%	294	119	117	200	101	100

Table 3: GC performance with and without root set scan

	Conservative		Exact	
Threads	Root Scan*	GC*	Root Scan*	GC*
1	160	72,598	583	72,396
10	575	74,484	2,647	76,031

*measured in μs

4.4 Target Root Set Scanning

The root set scan on the target JOP board is initiated by the GC. For each thread the stack is examined one frame at a time. Each frame is associated with one method, which owns a stack map with the necessary information to scan the local variables and the stack for potential references. The 32-bit garbage collection information structure contains enough information such that the packed stack map can be unpacked and used to identify the references.

5. EXPERIMENTS

Experiments are conducted with two example programs, which are part of our distribution. One is the famous *Hello World* program and the other is a larger program[4] for this embedded system. Note, that both examples also include the necessary library classes. We compare the overhead of the stack maps for these two programs.

Table 2 displays a comparison of the memory usage of the stack map in 32-bit words for the two Java programs `test.Hello` and `jbe.DoAll`. We compare all stack mapping algorithms against the best performing and set the index to 100 for the best performing algorithm. The experiments have been performed for a situation where all (”All” in the table) methods have its PCs mapped to a bit map. It has also been done for a simple reduction (”Only Ref.” in the table) where just the methods with at least one reference for some value of the PC are bit mapped. For each of these two approaches we investigate three different ways of packing the bit maps. The first packing method (”Raw”) is one bit record of the length for all locals and the maximum size of the operand stack for each PC. The second method (”Indx.”) is mimicking the constant pool and uses an index for each PC to map into the unique bit patterns for every method. The index uses no more bits than necessary and can vary from method to method. In addition, the unique patterns have the length of the sum of number of local variables and the maximum number of stack operands. Finally, the last way (”R.Indx.”) of packing is a small optimization over ”Indx.”. It reduces the information because it does not map the local or stack map bits if they are zero for all method PCs.

It is clear that the indexed approach (”Indx.” and ”R.Indx.”) is better than the ”Raw” approach. Furthermore, the obvious idea of not mapping bit maps for methods which never use a reference as local or operand are also good. But the last optimization of excluding the local or stack bits from the unique patterns if all are only 0 does not result in much memory usage reduction. It also adds a little more code on the target to decode the bit maps, so we conclude that the combination of ”Only Ref.” filtering and ”Indx.” packing is the most useful packing approach of those analyzed here.

5.1 Runtime Performance

The experiments are conducted with JOP running on an Altera Cyclone EP1C6Q240C8 FPGA [2] clocked at 80 MHz with a memory access time of 2 cycles. Our primary interest and goal with the experiment is to measure the overhead of the root scan process and compare it to the total collection process time. Accordingly, we have timed the root scan process (ie. a `synchronized` block) and the time to complete a `GC.gc()` invocation. The static references are grouped into one block in memory and this block of references is also scanned each time the GC process is initiated. Accordingly it is included in the timing experiments.

A potential disadvantage of the exact root set scan is the overhead that it adds to the garbage collection process. An advantage of using stack maps like we do here is that no overhead is encountered during execution and allocation, as it is only during garbage collection the exact root set scan is invoked to scan the stacks of all threads. In order to isolate the delay effects of the context switch effects on the root set scan itself we allocate no additional objects with reference type variables. If we allocated too many objects, perhaps in deeply linked structures then, we would get indications of the performance of the object traversal speed of the garbage collector which is not the aim here. Accordingly, we start a number of threads and then measure the time it takes to run the garbage collector with and without the exact root set scanner. These threads do nothing except wake up every two seconds but they still have a small stack of 15 words.

Column 1 in Table 3 shows the number of threads, column 2 shows the execution time in μs for the conservative root scan itself, column 3 shows the time for the total conservative GC process, forth column shows the execution time for the exact root scan, and the fifth column shows the time for the total exact GC process. The root set scan is inside an interrupt disabled code area and therefore we measure the time that the GC thread blocks other real-time threads for root scanning. Table 3 shows that the root scan overhead is low for a low number of threads. For one thread it even yields a total GC cycle time that is lower than the conservative scanning. The results are encouraging as the tradeoff for the longer blocking time is offset by the potential advantage of an exact root scan.

6. DISCUSSION

The discussion of the exact root set scan and the associated stack map information emphasize the flexibility of the system in a real-time multi-threaded environment.

The precise root set scan approach has some advantages over the conservative root set scan. Namely that the execution environment retains a predictable state as there is no risk that a set of primitives that look like potential object references interferes with the garbage collector. As JOP is a hard real-time JVM execution environment, we do not allow that any thread does not return control to the scheduler outside its pre-allocated period.

The root set scan process can be tailored toward the scheduling of the real time garbage collector. There are two cases we can discuss here:

[4]A collection of benchmarks for embedded Java systems named `jbe`.

- GC thread has lowest priority
- GC thread does not have lowest priority

The GC collection and the associated root set scan is convenient when the GC thread is assigned the lowest priority. That guarantees that it will not preempt any of the higher priority threads, which in turn ensures these threads' stack frames all have a PC the points to the instruction following the `invoke`. More specifically, the higher priority threads will have a top stack frame belonging to the method `waitForNextPeriod()` as the top frame. In this way, we now have a set of so-called *gcpoints*, which are specific places in the code that GC can take place. It means that we will be able to reduce the overhead of the full stack map that is shown in Table 2.

6.1 GC Scheduling

In [32] we argued that a periodic scheduled garbage collector can keep up with the demands of real-time threads. The GC is scheduled like any application thread. As the period of the GC thread is the longest and the deadline is the period the GC thread gets the lowest priority. With rate monotonic or deadline monotonic scheduling that means the GC thread will never preempt an application thread.

The application threads release the processor at known points (e.g. in [6] with `waitForNextPeriod()`). These points in the application are similar to GC preemption points [23]. That means that we exactly know all possible thread states when the root set scanning is performed. Knowing these thread states means knowing which methods can be on the stack for each thread. This knowledge has two implications:

1. We can omit the stack map information for all methods that can never be on the stack
2. The maximum stack height for each thread is known and we can bound the execution time of the root scanning

In summary, in the event that the GC is not assigned the lowest priority we are back in a situation that needs full stack map information. Because now the GC thread can preempt an executing thread at any valid value of the PC. This configuration can be useful for an application that executes for a long time and the low priority is used to ensure that it takes CPU cycles when none of the higher priority threads are executing. It is possible to avoid the full stack maps for code that is executed only by the threads with higher priority than the GC thread. For methods that are executed by threads with a lower priority than the GC thread we still need the full stack map.

6.2 WCET Analysis of Stack Walker

This section presents an analysis for illustrative purposes of the `wcet.gc.GCStkWalk.swk` method. This method scans the stack of a thread for references using the pre-compiled stack maps. Each method frame on the stack is inspected for references which are subsequently used in `com.jopdesign.sys.GC.gc` to get the exact root set. In order to schedule the garbage collector thread such that we are guaranteed completion within a given period, it is necessary to analyze it with respect to WCET [33]. Here we limit the analysis to the `swk` method itself. If we cast the problem in an integer linear programming (ILP) setting, it becomes possible to determine the WCET of the `swk` method [27]. The WCET for the `swk` is 35,235 machine cycles for one stack frame scan and an additional 34,797 cycles for each additional frame on the stack. For example, if the call tree has a depth of 5 methods then the stack walker would take approx. 175,000 cycles. These example numbers are obtained from running the `wcet.StartGCStkWlk` benchmark.

7. CONCLUSION AND FUTURE WORK

In this paper we presented an efficient way to find references in Java thread stack frames exactly. Due to features of the Java processor, we used for our implementation, that a thread can only be interrupted at bytecode boundaries it is possible to find all root references in the static fields and the stack frames. No internal processor registers have to be considered.

The presented solution for the problem of finding exact roots in the stack frames collects information about the stack layout from the class file during link time. Each method is annotated with information about the stack layout for each address. This stack map information can be retrieved by indexing the table with the program counter (which is part of the stack frame).

The organization as a table for each possible program counter table constitutes some memory overhead, but the access to this information is done in constant time. Therefore, this operation is WCET analyzable. To reduce this memory overhead we have presented two ways to pack this information. We achieved an reduction by about 60% compared to the raw information. The access to the packed information is still in constant time.

As future work we consider changing the Java processor to implement a typed stack. Practically this adds one bit to the 32-bit on-chip stack. The main issue for this implementation is the storage of this 33^{rd} bit in main memory when the on-chip stack gets exchanged with the main memory. With this hardware implementation of a typed stack we can compare it against the solution presented in this paper.

We do not implement the scheduling, but the reduction of the stack map due to the knowledge can be valuable. Finally, at present we scan all thread stacks atomically; in future work we plan to work on incremental root scanning to reduce blocking time.

8. REFERENCES

[1] O. Agesen, D. Detlefs, and J. E. Moss. Garbage collection and local variable type-precision and liveness in java virtual machines. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 269–279, New York, NY, USA, 1998. ACM Press.

[2] Altera. Cyclone FPGA Family Data Sheet, ver. 1.2, April 2003.

[3] C. Artho and A. Biere. Subroutine inlining and bytecode abstraction to simplify static and dynamic analysis. *Electronic Notes in Theoretical Computer Science*, 141(1):109–128, December 2005.

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

[5] H. G. Baker. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Not.*, 27(3):66–70, 1992.

[6] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[7] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[8] J. Cohen and A. Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Trans. Program. Lang. Syst.*, 5(4):532–553, 1983.

[9] M. Dahm. Byte code engineering with the BCEL API. Technical report, Freie Universitat Berlin, April 2001.

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

[11] A. Diwan, E. Moss, and R. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 273–282, San Francisco, CA, June 1992.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.

[13] F. Gruian and Z. Salcic. Designing a concurrent hardware garbage collector for small embedded systems. In *Proceedings of Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC 2005*, pages 281–294. Springer-Verlag GmbH, October 2005.

[14] T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J.-P. Lesot, and F. Parain. Memory management for real-time Java: an efficient solution using hardware support. *Real-Time Systems Journal*, 2002.

[15] A. Ive. Towards an embedded real-time java virtual machine. Licentiate thesis, Lund Institute of Technology, 2003.

[16] R. E. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[17] N. K., C. L., and R. M. Requirements for real-time extensions for the Java platform. Available at http://www.nist.gov/rt-java/, September 1999.

[18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[19] J. L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[20] A. F. Niessner and E. G. Benowitz. RTSJ memory areas and their affects on the performance of a flight-like attitude control system. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), LNCS*, 2003.

[21] K. D. Nilsen, S. Mitra, and S. J. Lee. Method for efficient soft real-time execution of portable byte code computer programs. United States Patent 6081665, 2000.

[22] A. Nilsson. Compiling java for real-time systems. Licentiate thesis, Dept. of Computer Science, Lund University, May 2004.

[23] A. Nilsson and S. G. Robertz. On real-time performance of ahead-of-time compiled java. In *ISORC*, pages 372–381, 2005.

[24] J. M. O'Connor and M. Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.

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

[26] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time java scoped memory: Design patterns and semantics. In *ISORC*, pages 101–110, 2004.

[27] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[28] S. G. Robertz and R. Henriksson. Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 93–102, New York, NY, USA, 2003. ACM Press.

[29] W. J. Schmidt and K. D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 76–85, New York, NY, USA, 1994. ACM Press.

[30] M. Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

[31] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[32] M. Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006.

[33] M. Schoeberl and R. Pedersen. Submitted to jtres 2006: Wcet analysis for a java processor. 2006.

[34] F. Siebert. Real-time garbage collection in multi-threaded systems on a single processor. In *20th IEEE Real-Time Systems Symposium (RTSS'99)*, Phoenix, Arizona, 1999.

[35] G. L. Steele. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, 1975.

[36] P. R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, Jan. 1994. Expanded version of the IWMM92 paper.