Gain Time Reclaiming in High Performance Real-Time Java Systems

Erik Yu-Shing Hu[*], Andy Wellings and Guillem Bernat

Real-Time Systems Research Group
Department of Computer Science
University of York, York, YO105DD, UK
{erik,andy,bernat}@cs.york.ac.uk

Abstract

The run-time characteristics of Java, such as high frequency of method invocation, dynamic dispatching and dynamic loading, make Java more difficult than other object-oriented programming languages, such as C++, for conducting Worst-Case Execution Time *(WCET) analysis. To offer a more flexible way to develop object-oriented real-time applications in the real-time Java environment without loss of predicability and performance, we propose a novel gain time reclaiming framework integrated with WCET analysis. This paper demonstrates how to improve the utilisation and performance of the whole system by reclaiming gain time at run-time. Our approach shows that integrating WCET with gain time reclaiming can not only provide a more flexible environment, but it also does not necessarily result in unsafe or unpredictable timing behaviour.*

Keywords : Gain Time Reclaiming, Worst-Case Execution Time (WCET) Analysis, Real-Time Java

1 Introduction

There is a trend towards using object-oriented programming languages, such as Java and C++, to develop real-time systems because the use of such languages has several advantages, for instance reusability, data accessibility and maintainability. The success of hard real-time systems, undoubtedly, relies upon their capability of producing functionally correct results within defined timing constraints. In order to achieve this, the processor and resource requirements of the hard real-time tasks have to be reserved. However, this may result in under utilisation and lead to very poor performance for aperiodic tasks. Unfortunately, object-oriented programming languages support more dynamic behaviour than procedural programming languages, and some of these features may result in object-oriented applications having a more pessimistic worst-case behaviour. In consequence, object-oriented real-time systems may suffer from significantly lower utilisation and poorer overall performance of the whole system than procedural real-time systems.

Most scheduling algorithms assume that the Worst-Case Execution Time (WCET) estimation of each task is known prior to conducting the schedulability analysis. Typically, the WCET analysis and schedulability analysis are carried out separately. Sophisticated techniques [6, 19, 21], are used in WCET analysis, for instance to model caches and pipelines, to achieve safe and tight estimation. However, most WCET analysis approaches are only considered in relation to procedural programming languages. Performing analysis on object-oriented programs must take into account additional dynamic features, such as dynamic dispatching and memory management. Some research groups have proposed various approaches [11, 20] to address these issues, but most approaches result in development environments which are inflexible and very limited.

In contrast with the WCET analysis, a number of research groups have proposed various flexible scheduling algorithms, for instance priority server algorithms [5] and a slack stealing algorithm [18], to provide a more flexible real-time development environment with greater performance of the whole system. In general, these flexible scheduling algorithms are mainly focused on improving the performance of the aperiodic tasks at run-time. They have, however, paid insufficient attention to the fact that, for the most part, hard real-time tasks are not executing via the worst-case execution time path. Therefore, even though they have demonstrated very complex scheduling algorithms to improve the average performance of the whole system, the improvements are still limited and the overhead of the implementation is extremely high or it is sometimes not even possible to implement them in practice.

Generally, the spare capacity of a real-time system may be divided into three groups [8]: *extra capacity*, *gain time*, and *spare time*. Extra capacity is the capacity which is not allocated for hard real-time tasks during the design phase. This can be identified off-line. Gain time is produced when the hard real-time tasks execute in less than their worst-case execution time estimations. This may only be reclaimed at run-time since it depends on the actual executions of tasks [8]. Spare time may be defined as a situation in which sporadic tasks do not arrive at their maximum rate. Most flexible scheduling algorithms are mainly focused on reclaiming the extra capacity of the system. Only a few research approaches [1, 9, 13] have discussed how to reclaim gain time. Even here, they have tended to focus on procedural programming languages, rather than on object-oriented programming languages.

This paper introduces a novel gain time reclaiming frame-

[*]This work has been funded by the U.K. EPSRC under Grant GR/M94113.

work integrated with WCET analysis to balance the tradeoff among flexibility, efficiency and predictability. In our approach, the predictability of hard real-time tasks is strengthened during the design phase and the performance of the whole system is reinforced with gain time reclaiming during run-time. We show that integrating WCET analysis with gain time reclaiming not only may achieve high utilisation and high performance of the whole real-time system, but also keeps the flexibility and reusability of the object-oriented real-time applications. The major contributions of this paper are:

- introducing a novel gain time reclaiming framework integrated with WCET analysis for real-time Java applications,
- demonstrating how to reclaim the gain time of object-oriented real-time systems with gain time reclaiming graphs, and
- constructing a bridge between WCET analysis and scheduling algorithms to provide greater flexibility without loss of predicability and efficiency.

The rest of the paper is organised as follows. Section 2 gives a brief review of related work, while Section 3 presents an overview of our previous work. Section 4 demonstrates how gain time can be reclaimed in high performance object-oriented real-time systems. Then, Section 5 gives an overview of implementation issues in the real-time Java environment and Section 6 evaluates our approach with a practical example. Finally, conclusions and future work are presented in Section 7.

2 Related Work

This section gives a brief survey of the related work on gain time analysis [1, 9, 13]. Haban and Shin have proposed an approach [13] placing software triggers at the end of basic blocks[1] in task code to measure actual execution time. By comparing the actual execution time calculated at the software trigger point with pre-determined WCET values, the gain time of the specific basic block can be determined. In a similar way, Dix et al. have proposed an approach [9] adding *milestones* into task code to calculate the maximum remaining execution time of the particular task. However, both approaches reclaim the gain time after they have been generated and are not integrated with WCET analysis.

Audsley et al. [1] have introduced a *gain point* mechanism to reclaim gain time of the basic blocks of a task code as early as possible. In [1], the use of gain point can be grouped into four separate forms, including *static gain point* for static code, *dynamic gain point* for loop constructs, *efficiency gain point* for detecting hardware speed-ups, and *resource usage gain point* for identifying spare resources. Yet, Audsley et al.'s approach and the previous two approaches do not take into account object-oriented programming features or the overestimated WCET values resulting from functional constraints.

[1]A basic block is a continuous section of code in the sense that control flow only goes in at the first instruction and leaves through the last one.

3 Previous Work

We have proposed an extensible distributed high-integrity real-time Java environment [16], called XRTJ, which extends the Real-Time Java architecture [4] proposed by the Real-Time Java Expert Group. In our environment, the *Extensible Annotations Class* (XAC) format[2][15] is used to store extra information that cannot be expressed in the source code. We have also demonstrated how the dynamic dispatching issues in WCET analysis can be addressed with minimum annotations to estimate safe and tight WCET bounds for hard real-time applications in [14].

3.1 Portable WCET Analysis

A portable WCET analysis approach based on the Java architecture has been proposed by Bernat et al. [3, 2]. This section presents how portable WCET analysis can be adapted for our environment [16].

Portable WCET analysis uses a three-step approach: high-level analysis, which analyses the annotated Java class files and computing the portable WCET information in the form of *Worst-Case Execution Frequency* (WCEF) vectors [2], representing execution-frequency information about basic blocks and more complex code structures that have been collapsed during the first part of the portable WCET analysis; low-level analysis, which produces a *Virtual Machine Time Model* (VMTM) for the target platform by performing platform-dependent analysis on Java bytecode instructions implemented for the particular platform; and conducting the combination of the high-level analysis with the low-level analysis to compute the actual WCET bound of the analysed code sections.

In our environment, an annotation-aware Java Compiler analyses the annotated Java programs and creates the WCEF vectors during compilation. The WCET vectors and WCET annotations are stored in the XAC file. Therefore, after compilation, the class files and XAC files are ready for WCET analysis tools. A WCET analysis tool then performs the combination of the high-level analysis with the low level VMTM to compute the actual WCET bound of the analysed code sections.

3.2 WCET Annotation

In [14], we have introduced the `//@maxWCET()` annotation to address complicated class hierarchies where there is more than one particular method that may be invoked. It can suggest that the WCET of a dispatching method should be considered to be the maximum WCET of the class family[3] containing that method. In this annotation, "&" may be applied to denote the whole class family of a class/interface, whereas "+" and "-" can also be used to express the union or subtraction of a single class/interface or a class family for the method respectively.

[2]The XAC format is an annotation structure that can be stored in files or as an additional code attribute in *Java Class Files* (JCF).

[3]A class family of a class/interface is a set of the classes and/or interfaces including the class/interface itself and all the child classes and/or interfaces inherited from it.

4 Gain Time Reclaiming

Given the need to provide 100% deadline predictability for hard real-time tasks, it is inevitable that the processor and other resources will be under-utilised at run-time [1]. In addition, to avoid unpredictable behaviour in hard real-time applications, some dynamic features of the object-oriented programming are often prohibited from being used [12]. As a result, it is likely that the design of object-oriented real-time systems could become inflexible and the performance and utilisation relatively poor. In order to balance the tradeoff among the flexibility, predictability and efficiency of the real-time systems, a novel gain time reclaiming framework is proposed.

From the point of view of the syntax of the programming languages, the levels of flow of control can be classified as follows [10]: *local flow of control*, which identifies statements within a routine or method to be executed; *method invocations and routine calls*, performing the parameter transfer and flow-of-control manipulation needed to activate a new routine; and *non-local jumps*, which divert the control flow from the currently running routine into an ancestor routine. In general, real-time threads are not allowed to use non-local jumps since this may result in unpredictable and unanalysable behaviour. Therefore, gain time reclaiming in this paper will be focused on the first two levels of flow of control in object-oriented programming languages. Integrating these levels with WCET analysis techniques, gain time reclaiming in object-oriented programming languages may be classified into three mechanisms: *structural constraint reclaiming* (§ 4.1), *object constraint reclaiming* (§ 4.2), and *functional constraint reclaiming* (§ 4.3).

Note that gain time can be represented with machine cycles of the target machine if the source code of the application is translated into machine code directly. However, it could be difficult to estimate the exact machine cycles of Java applications because of the portability of Java architecture. In this case, the concept of WCEF vectors [2] can be used instead of pre-calculated units. These WCEF vectors may be used to calculate the exact gain time when the information about the target machine is available. Machine cycle units are used in the rest of the paper for the take of clarity.

4.1 Structural Constraint Reclaiming

On the whole, a local flow of control is made up of a number of basic blocks in the form of sequences, *selections* (i.e. some pieces of code to be selected for execution based on the value of some expressions) and *repetitions* (i.e. pieces of code to be executed zero or more times based on the value of some expressions). Therefore, the overestimated WCET bounds which suffer from the structure of the program can be reclaimed as soon as the exact execution path of selection code or the exact number of iterations of repetition code are determined. Formally, based on the WCET analysis rules defined in Timing Schema [22], the gain time of the structural constraints can be defined as follows:

- Let S be a selection code with the expression Z, and let P be an actual executed path of S in a particular execution. Then, the gain time of P can be calculated by subtracting the sum of the WCET of Z and the WCET of P from the WCET of S. This schema can be used in any type of selection code, such as `if-then-else` and `switch-case`.
- Consider a repetition code R with the expression Z and loop X. Here, assume that n is the maximum loop bound of R used in the static WCET analysis and n' is an actual executed iteration in a specific execution. Then, the gain time of n' iteration of R can be computed by multiplying the subtraction of n' from n by the sum of the WCET of Z and the WCET of X. This schema is valid for any type of repetition code with a bounded number of iterations, such as `for-loop, while-loop` and `do-while`.

The structural constraint reclaiming of a specific thread may be represented with a *Structural Gain Time Reclaiming Graph* (SGTRG), which illustrates the exact places (i.e. offset number of the machine code or Java bytecode) and amounts (i.e. machine cycles or WCEF vectors) of gain time that may be reclaimed. Formally, the SGTRG can be denoted with *gain time reclaiming nodes* $\nabla(l, g)$, where l represents the offset number of the Java bytecode and g indicates the amounts of gain time that can be reclaimed.

Using compiler techniques [10] that are applied to the analysis of the local flow of control can identify the exact places of the basic blocks of the selection and repetition code, and derive an SGTRG for each routine or method. Based on the SGTRG, the gain time of structural constraints can be reclaimed by determining the actual execution path of selection code or the exact iteration times of repetition code at run-time.

```
1  public check_data () {
2    int i , morecheck, wrongone;
3    i =0; morecheck=1; wrongone=−1;
4     ...
5    while (morecheck) {
6       ...
7      /* Say WCET(if) = 70 cycles
8               WCET(else) = 100 cycles . */
9      if (data[i] < 0) {
10     /* Here 30cycles of the structural
11     gain time of the current if−else
12     statement can be reclaimed.
13     (i.e. 100−70 cycles) */
14        ...
15       wrongone=i; morecheck=0;
16     }
17     else {
18     /* Here 50 cycles of funtional
19     gain time of the below if−else
20     statement can be reclaimed.
21     (i.e 100−50 cycles) */
22          ...
23         if (++i >= DATASIZE)
24           morecheck=0;
25       }
26   }
27    ...
28   /* Say WCET(if) = 100 cycles
29            WCET(else) = 50cycles. */
30   if (wrongone >= 0) {
31     // Error path ;
32      ...
33     return 0;
34   }
35   else {
36     // Noml path ;
37      ...
38     return 1;
39   }
40 }
41 ...
```

Figure 1. An example of gain time reclaiming [19]

As shown in Figure 1, the `if-then-else` basic block can reclaim 30 cycles at Line 10, if the condition expression is TRUE (i.e. `data[i]<0`) and the `while-loop` is part of its worst-case path. One should note that the gain time reclaiming for unknown repetition needs to be provided with the maximum loop bound used in the WCET analysis. The maximum iteration number may be provided by manual annotations or derived automatically using Gustafsson's approach [11]. For example, the `while-loop` (Line 5-26), given in Figure 1, needs to be provided with such information to be able to reclaim the gain time of the repetition code at run-time.

4.2 Object Constraint Reclaiming

In order to guarantee all hard real-time tasks in object-oriented real-time systems, most WCET researchers suggest prohibiting the use of dynamic dispatching, dynamic loading and garbage collection features [12]. We have argued for the need to use dynamic dispatching and demonstrated how to guarantee the deadline of hard real-time tasks in our previous work [14]. In our previous approach, a `//@maxWCET()` annotation is used to indicate the WCET of a dynamic dispatching method call. However, we cannot avoid the fact that the use of `//@maxWCET()` might have relatively pessimistic results if the class family is extremely large or the WCET estimations for different classes are spread over a wide range. In order to compensate for the penalty of the flexibility of the object-oriented programming, object gain time reclaiming is required. Formally the gain time of the object constraints can be defined as follows:

- Consider an object X with a dynamic dispatching method m, and assume that the WCET of $X.m$ should take into account the WCET of the class family F. Let i be an instance of X and A be an actual type of the instance to be executed in a particular execution. Then, the gain time of $i.m$ in the particular execution can be calculated by subtracting the WCET of $A.m$ from the WCET of $F.m$.

Based on the thread-based CFG[4], all instances of various objects that are created in each real-time thread can be identified. Then, by analysing the assignment or type changing code of each instance we can distinguish the lifetimes of particular types of each instance, and produce an *Object Type Lifetime Graph* (OTLG), a diagram which represents the lifetimes of types of particular instances in a specific thread. Formally, an OTLG is made up of two types of component: *node* and *edge*. A *node*, named *type changing node* in Figure 2, denotes a place where the type of instance is changed, whereas an *edge* illustrates the lifetime of a particular type of instance between two nodes. An OTLG can be represented with a number type changing nodes that can be formally expressed with $\Theta(l, t)$, where l indicates the offset number of the Java bytecode and t denotes the possible types of the instance at run-time.

In the OTLG, symbolic type references may be applied to represent the relationship between the dynamic dispatching objects of the same class family during run-time. After discriminating the lifetimes of specific types of each instance, analysing method invocations on each type of the instance can determine the amount of gain time that can be reclaimed. Here, symbolic references that are represented in OTLGs can be solved by analysing the associated instances in the specific thread incrementally so that gain time can be reclaimed as early as possible. Following this, the exact places and amounts of object gain time reclaiming can be identified and illustrated in an *Object Gain Time Reclaiming Graph* (OGTRG). An *OGTRG* is a diagram which illustrates places where the type of the instance should be traced or the object constraint reclaiming may take place.

[4]A thread-based CFG is a control flow graph which illustrates not only the local flow of control of the thread, but also the local flow of control of each method invocation of the thread in detail.

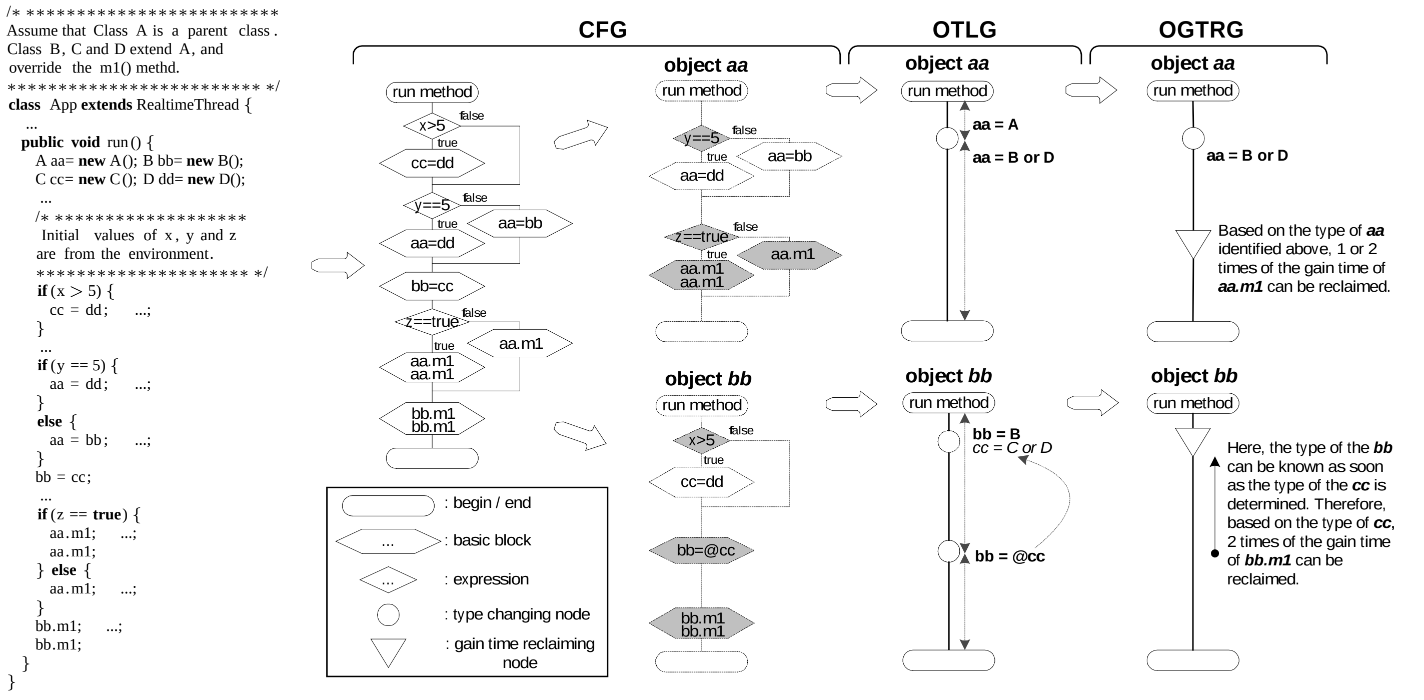
Formally, an OGTRG is made up of two types of nodes: *type changing node* and *gain time reclaiming node*. A *type changing node* denotes a place where the type of instance is changed but where it is not possible to identify the exact amount of gain time, whereas a *gain time reclaiming node* indicates a place where the exact amount of the gain time of a particular type of the instance occurs. As mentioned above, type changing nodes and gain time reclaiming nodes can also be denoted with $\Theta(l, t)$ and $\nabla(l, g)$ in an OGTRG respectively. The gain time reclaiming of all instances in the real-time task can be merged together and provided for the run-time environment to reclaim them.

Considering the example in Figure 2, four instances (i.e. `aa,bb,cc,` and `dd`) need to be analysed to carry out the object constraint reclaiming analysis in the `App` real-time thread. Using the object constraint reclaiming mechanism mentioned above, an object gain time reclaiming graph for each instance can be conducted by a modified compiler or tool automatically. A diagram which illustrates the transformation of two instances (i.e. `aa` and `bb`) from CFG to OGTRG is given in Figure 2. In the figure, the type of instance `aa` can be identified in the second `if-then-else` statement (i.e. type changing node) and the exact number of method invocations can be determined in the last `if-then-else` statement (i.e. gain time reclaiming node). Therefore, as soon as the expression of the last `if-then-else` statement is executed, the object gain time of instance `aa` can be reclaimed.

Note that solving the symbolic expression of an associated class family can improve the reclaiming as early as possible. As shown in Figure 2, the gain time of the instance `bb` can be reclaimed as soon as the type of instance `cc` is determined. Therefore, the gain time reclaiming node of the instance `bb` can be indicated at the first `if-then-else` statement. Our approach can also be applied to the analysis of method-based applications that are built as libraries or packages in object-oriented programming.

4.3 Functional Constraint Reclaiming

Identifying the *exclusive paths* [19] or various *modes* [6], based on design knowledge, to calculate the WCET estimations of the real-time applications has been widely used in the WCET field. By analysing real-time threads with annotations that indicate exclusive paths or modes, relatively safe and tight WCET bounds can be estimated. We acknowledge such efforts and contributions in the WCET domain. However, one should note that it is possible that the WCET estimations of the exclusive paths or different modes are spread over a wide range, and the exact execution path or mode cannot be determined during the design phase. In such cases, the reductions of the pessimistic estimations can still be limited. Moreover, analysing functional



```
/* *************************
Assume that Class A is a parent class.
Class B, C and D extend A, and
override the m1() methd.
************************* */
class App extends RealtimeThread {
  ...
  public void run() {
    A aa= new A(); B bb= new B();
    C cc= new C(); D dd= new D();
    ...
    /* *******************
     Initial values of x, y and z
    are from the environment.
    ********************* */
    if (x > 5) {
      cc = dd;     ...;
    }
    ...
    if (y == 5) {
      aa = dd;     ...;
    }
    else {
      aa = bb;     ...;
    }
    bb = cc;
    ...
    if (z == true) {
      aa.m1;    ...;
      aa.m1;
    } else {
      aa.m1;    ...;
    }
    bb.m1;    ...;
    bb.m1;
  }
}
```

CFG
OTLG
OGTRG
run method
x>5
false
true
cc=dd
y==5
aa=bb
aa=dd
bb=cc
z==true
aa.m1
aa.m1 aa.m1
bb.m1 bb.m1
object aa
object bb
bb=@cc
aa = A
aa = B or D
bb = B
cc = C or D
bb = @cc
Based on the type of aa identified above, 1 or 2 times of the gain time of aa.m1 can be reclaimed.
Here, the type of the bb can be known as soon as the type of the cc is determined. Therefore, based on the type of cc, 2 times of the gain time of bb.m1 can be reclaimed.
: begin / end
: basic block
: expression
: type changing node
: gain time reclaiming node

Figure 2. An example of object constraints and its diagram of producing OGTRG

constraints in object-oriented programming is much more complicated than in procedural programming. This means that data dependency issues in object-oriented programming langues can lead to pessimistic WCET estimations that suffer from not only structural constraints but also object constraints. Hence, as well as analysing the exclusive paths and modes in the structure of the programs, the analysis of functional constraints must take into account the pessimistic bounds that suffer from the dynamic characteristic of object types in object-oriented programming.

For the most part, these pessimistic WCET estimations that suffer from data dependencies can be addressed with the structural and object constraint reclaiming mechanisms. However, to be able to reclaim gain time as early as possible, these mechanisms can be integrated with the WCET analysis approaches that bear data dependencies issues in mind.

The rationale of functional constraint reclaiming is based on the concept of identifying the exclusive paths or various modes of the WCET analysis approaches. Therefore, we assume that the exclusive paths or various modes can be distinguished with annotations that are introduced by WCET analysis. Taking advantage of these annotations, a modified compiler or gain time reclaiming tool can derive these exclusive (or inclusive) paths and modes from the programs. Based on the identified paths and modes, we can determine the places where these paths or modes can be distinguished. As soon as the exact path or mode can be discriminated or executed, the gain time of the path or mode can be reclaimed. Here, the gain time of particular paths can be reclaimed if the exclusive path of the current executing path is the worst-case execution path or the inclusive path of the current path is not the worst-case execution path. Similarly, as soon as the specific mode is identified and it is not the worst-case execution mode, then the gain time of the mode can be reclaimed. In both cases, the functional constraint reclaiming should consider the data dependencies issues connected with object constraints. Formally, the gain time of the functional constraints can be defined as follows:

- Suppose that S is a section of code that includes exclusive paths, and P is an actual executed path of S in a particular execution. Then, the gain time of P can be computed by subtracting the WCET of P from the WCET of S. This schema can also be used in calculating exclusive modes of functional constraints.

The functional constraint reclaiming of a specific thread may be represented with a *Functional Gain Time Reclaiming Graph* (FGTRG), which illustrates the exact places and amounts of gain time that may be reclaimed. The functional gain time reclaiming can be assumed as an advanced mechanism that is a combination of the structural and object constraint reclaiming mechanisms of a particular path or mode. Therefore, the formal definitions of the nodes introduced for OGTRG (i.e. $\Theta(l, t)$ and $\nabla(l, g)$) can also be applied to FGTRG.

For example in Figure 1, the WCET estimations of two `if-then-else` statements (i.e. Line 9-25 and Line 30-39) are exclusive to each other, but the structure of the program may increase the WCET estimation. In other words, the WCET of these two `if-then-else` statements will not be executed at the same time due to its functional constraints. Therefore, based on WCET annotations identifying the exclusive paths or modes, the gain time associated with the normal mode (i.e. the `else` statement at Line 35-39) can be reclaimed in Line 18 at run-time. It can be observed that using functional constraint reclaiming may reclaim the gain time earlier than the structural constraint reclaiming.

5 Implementation

We use the XRTJ architecture [16] to demonstrate how the gain time reclaiming can be implemented for object-oriented programs. However, the implementation approach is not restricted to our architecture. Following the philosophy of portable WCET analysis, the implementation may be divided into two stages: *producing gain time reclaiming graphs* that can be conducted at compilation, and *run-time supported reclaiming* that can be carried out at the target platform.

Here, we assume that the WCET values or WCEF vectors of each basic block have been calculated. They may be produced while performing static timing analysis by combining the portable WCET analysis [2, 3] and our previous approach [14, 15]. Then the WCET value or WCET vector of each basic block can be stored in the XAC format in either separate files or code attributes in JCF files. Note also that the gain time reclaiming graphs can be provided in the XAC format. In addition, the worst-case execution for the runtime overhead of the gain time reclaiming can be added into the basic block where the gain time reclaiming takes place.

5.1 Producing Gain Time Reclaiming Graphs

Based on the thread-based CFG of each real-time thread, gain time reclaiming graphs can be produced from analysable source programs or JCF files by a modified compiler or tool that supports our gain time reclaiming mechanisms discussed above. These gain time reclaiming graphs can be extracted during compilation in order to reduce the run-time overhead. The three gain time reclaiming graphs of a particular thread can be merged together into the the *Thread Gain Time Reclaiming Graph* (TGTRG), which illustrates places where the gain time reclaiming may take place in the thread. It should be noted that gain time that can be reclaimed by functional constraint reclaiming could overlap with those that can be reclaimed by the structural and object constraint reclaiming. Therefore, combining the functional gain time reclaiming graph into TGTRG leads to the necessity of eliminating the gain time that overlap with other constraints. The implementation overhead is relatively low since the implementation does not necessarily need to know all possible execution paths in advance.

5.2 Run-Time Support Reclaiming

For the most part, the gain time reclaiming may be carried out at the target platform in two stages: *Reconstructing* and *Reclaiming*. The first stage, called *reconstructing*, includes mapping Java bytecode with associated gain time reclaiming nodes, and instrumenting gain time reclaiming method invocations. A modified Java virtual machine or tool loads the associated Java class files and the TGTRG of each task-based real-time thread stored in the XAC files, and reproduces the relationship between gain time reclaiming nodes and Java bytecode. Then, the gain time reclaiming graphs can be translated into method invocations that support the integration with the scheduling algorithms. Here, the exact machine cycles of gain time which are provided with WCET vectors can be calculated.

Note that it could be possible that the actual reclaimed gain time is less than the run-time overhead of the reclaiming. In this situation, the gain time should be either neglected or accumulated until it is worth reporting. Which gain time reclaiming nodes need to be removed can be identified on the basis of an acceptance value that may be used to examine if the overhead of the gain time reclaiming is acceptable at run-time. Furthermore, to support repetition code with gain time reclaiming mechanism, either the run-time system, such as the Virtual Machine, must support a mechanism to count the exact iteration of the loop at run-time or additional code must be introduced by a modified compiler to count the loops. In addition, type tracing mechanism need to be provided for type changing nodes where the exact amount of the gain time cannot be identified.

At the reclaiming stage, based on gain time reclaiming methods instrumented during the reconstructing stage, gain time can be reclaimed automatically as soon as reclaiming methods are executed. The gain time can be collected by the associated scheduling algorithm and can be used to improve the overall performance of the whole system. How to integrate the gain time reclaiming with the scheduling algorithm is outside the scope of this paper. Techniques such as Dual-Priority Scheduling [7] and Dynamic Sporadic Server [5] are applicable.

6 A Practical Example

Considering the "Rover Tasks" part of the applications layer of the FIDO Rover system[5] in Figure 3, the command processor sends discrete instructions to particular periodic tasks at run-time to interact with the environment. Therefore, any sequences of the commands may be sent by the command processor to a specific periodic task. In addition, it is possible that all instructions of the various types of robot cannot be known beforehand (i.e. during the design phase of the command processor). Consequently, if any new command is introduced or a new type of robot is developed, the command processor and the specific periodic task need to be revised and the whole system needs to be retested. Unfortunately, redevelopment and retesting are relatively expensive in high performance real-time applications.

In order to provide the applications with greater flexibility, reusability and extensibility, the application layer may be redeveloped for a real-time Java environment. Based on the application layer of the FIDO Rover's architecture, `Instructions` classes can be classified into a subclass family based on their similar functionalities or characteristics. Part of the class hierarchy of the `Instructions` classes is given in Figure 4. Any

[5]The FIDO (Field Integrated Design and Operations) Rover system is a planetary exploration autonomous system and is being used in ongoing NASA field tests to simulate driving conditions on Mars.

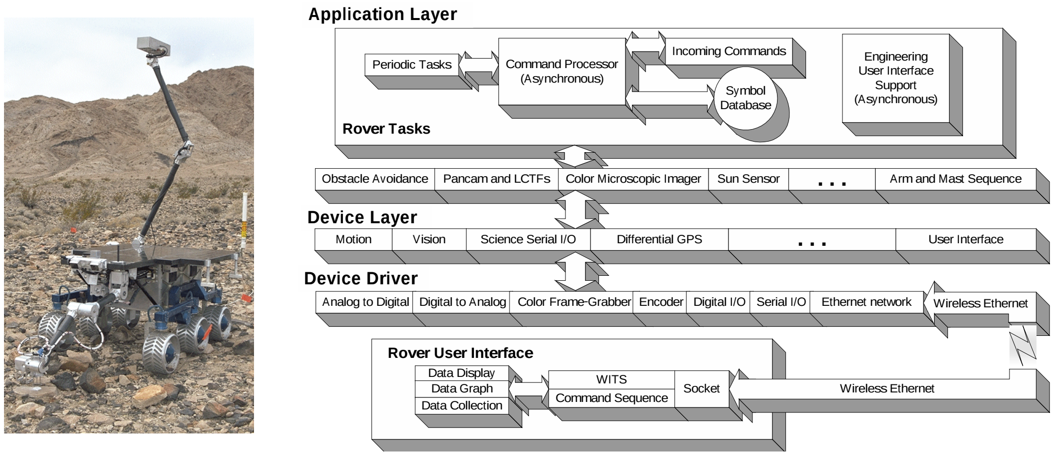


Figure 3. FIDO Rover prototype and its software implementation layers [17]

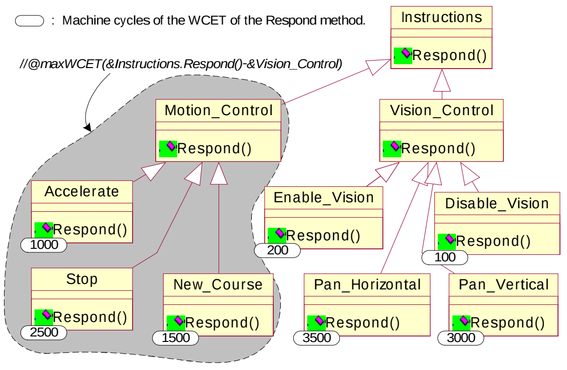


Figure 4. A class hierarchy of the `Instructions` Classes with WCET estimations

particular function or action of the child classes can be specified in their `Respond()` methods by overriding the method of the parent class. A typical periodic task that may be used in the FIDO Rover system is given in Figure 5. As shown in the figure, the only information which can be known during the design or analysis phase is the maximum number of instructions sent to the specific periodic task and the WCET estimation of each instruction. Undoubtedly, to be able to ensure that these critical tasks have to be finished by their deadline, all the requirements of the processor and resources need to be reserved. For the most part, the WCET estimation of each periodic task can be calculated with a maximum number of instructions allowed to be sent to the specific thread and the maximum WCET estimation of the instructions from the whole `Instructions` class family. In this situation, if the WCET estimations of all the instructions are spread over a wide range, the estimation is very pessimistic. In addition, the overall performance will be decreased because these reserved resources cannot be used for other aperiodic tasks. The next subsection (§ 6.1) explores how these issues can be addressed with our approach.

6.1 Analytical Results

Assume that the WCET values of each `Respond()` method of the `Instructions` class family are given as in Figure 4

```
 1 /*******************************************
 2 Motion Control  periodic  real—time thread
 3 ********************************************/
 4 import javax. realtime .*;
 5  ...
 6 public class MotionCtrl_RT extends RealtimeThread {
 7   public MotionCtrl_RT (...) {
 8       ...
 9   }
10    ...
11   // Maximum number of instructions allowed in each period
12    final  int  maxInsts = 10;
13   /*****************************************
14   Say ObjList object contains a list of Instrucions objects
15  ********************************/
16   private ObjList CommandList[maxInsts];
17    ...
18   public void run() {
19     while(!Terminate) {
20        // Do the main job of the periodic thread
21        ...
22        // Get the number of commands for this period.
23        ints = getNumberOfCommands();
24        ...
25        // Get the commands for this period.
26        CommandList = getCommandList();
27        for(i=0; i<ints; i++) {
28           ...
29           result = (CommandList[i]).Respond();
30           ...
31        }
32        waitForNextPeriod ();  // waiting for next period
33     }
34   }
35 }
```

Figure 5. An example of periodic thread

and the WCET estimation of the `run()` method of the rest of code in the `MotionCtrl_RT` thread is 150 cycles. In the `MotionCtrl_RT` object, the maximum number of instructions is 10 (at Line 12). Therefore, if all classes are taken into account for the estimation of the specific periodic of the `MotionCtrl_RT` thread, the WCET value is equal to 35150 cycles (i.e. 10*3500+150).

However, as shown in Figure 4, based on the design knowledge, the estimation for `MotionCtrl_RT` can be provided with `//@maxWCET()` annotation at Line 29, since the WCET estimations of the `Respond()` method in the `Vision_Control` class family are not going to be used in the `MotionCtr_RT` thread. Therefore, the estimation of the WCET value of the thread can be reduced to 25150 cycles (i.e. 10*2500+150). Then, based on the thread gain time reclaiming graph, including a structural constraint reclaiming node at

Line 23 and an object constraint reclaiming node at Line 26, the gain time of each period of the `MotionCtrl_RT` thread can be reclaimed at run-time. For example, if the context of the `CommandList` for the thread includes three instructions including `New_Course, Accelerate` and `Stop`, the gain time of the particular period can reclaim 20000 cycles (i.e. 25150-5150 cycles). In the same way, other periodic threads in the Rover tasks part of the application layer may be analysed in the design phase and reclaimed at run-time in order to improve the overall performance of the system.

7 Conclusions and Future Work

This paper has demonstrated a novel gain time reclaiming framework integrated with WCET analysis for high performance real-time Java systems. Our approach shows that integrating WCET with gain time reclaiming not only can provide a more flexible environment to develop real-time Java applications, but may also achieve high utilisation and high performance of the whole real-time system.

As discussed above, three types of gain time reclaiming mechanisms can be applied to real-time Java programs. The analysis of structural constraint reclaiming and object constraint reclaiming can be fully automatic with a modified compiler or tool that supports our mechanisms, whereas the functional constraint reclaiming partially needs to be integrated with annotations used in WCET analysis. Furthermore, these mechanisms can also be applied to the development of real-time Java API (Application Programming Interfaces) or packages to improve the utilisation and overall performance of the whole system.

Attention should be drawn to the fact that the granularity of the gain time reclaiming should be balanced to avoid large run-time overheads. Therefore, our future work will be focused on when gain time should be reclaimed or how to evaluate which gain time can be reclaimed without rendering the run-time environment inefficient.

References

[1] N. C. Audsley, R. I. Davis, and A. Burns. Mechanisms for Enhancing the Flexibility and Utility of Hard Real-Time Systems. *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 12–21, December 1994.

[2] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework. *Proceedings of the 6th IEEE Real-Time Computing Systems and Applications RTCSA-2000*, pages 39–48, December 2000.

[3] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. *Proceedings of the 6th Euromicro Conference on Real-Time Systems*, pages 81–88, June 2000.

[4] G. Bollella, J. Gosling, B. M. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *Real-Time Specification for Java*. Addison Wesley, 2000.

[5] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable scheduling algorithms and applications*. Kluwer Academic Publishers, 1997.

[6] R. Chapman, A. Burns, and A. Wellings. Integrated Program Proof and Worst-Case Timing Analysis of SPARK Ada. *Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[7] R. Davis and A. Wellings. Dual Priority Scheduling. *Proceedings of the 16th IEEE of Real-Time Systems Symposium*, pages 100–109, December 1995.

[8] R. I. Davis. *On Exploiting Spare Capacity in Hard Real-Time Systems*. Ph.d. thesis, Department of Computer Science, University of York, UK, July 1995.

[9] A. Dix, R. Stone, and H. Zedan. Design Issues for Reliable Time-Critical Systems. Technical Report YCS-133, Department of Computer Science, University of York, UK, 1990.

[10] D. Grune, H. E. Bal, C. J. Jacabos, and K. G. Langendoen. *Modern Compiler Design*. Wiley, 2000.

[11] J. Gustafsson. *Analysing Execution Time of Object-Oriented Programs with Abstract Interpretations*. Ph.d. thesis, Department of Computer Systems, Information Technology, Uppsala University, Sweden, May 2000.

[12] J. Gustafsson. Worst Case Execution Time analysis of Object-Oriented Programs. *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems WORDS-2002*, pages 58–63, January 2002.

[13] D. Haban and K. Shin. Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times. *IEEE Transactions on Software Engineering*, 16(12), December 1990.

[14] E. Y.-S. Hu, G. Bernat, and A. J. Wellings. Addressing Dynamic Dispatching Issues in WCET Analysis for Object-Oriented Hard Real-Time Systems. *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC-2002*, pages 109–116, April 2002.

[15] E. Y.-S. Hu, G. Bernat, and A. J. Wellings. A Static Timing Analysis Environment Using Java Architecture for Safety Critical Real-Time Systems. *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems WORDS-2002*, pages 77–84, January 2002.

[16] E. Y.-S. Hu, J. Kwon, and A. J. Wellings. XRTJ: An Extensible Distributed High-Integrity Real-Time Java Environment. *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications RTCSA-2003*, February 2003.

[17] Jet Propulsion Laboratory (JPL). Field Integrated Design & Operations (FIDO) rover. NASA. http://fido.jpl.nasa.gov/.

[18] J. Lehoczky and S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. *Proceedings of the 13th IEEE of Real-Time Systems Symposium*, pages 110–123, December 1992.

[19] Y. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. *ACM SIGPLAN Workshop on Language, Compilers and Tools for Real-Time Systems*, June 1995.

[20] P. Persson and G. Hedin. An Interactive Environment for Real-Time Software Development. *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe 2000)*, June 2000. St. Malo, France.

[21] P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.

[22] A. Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.