A Novel Gain Time Reclaiming Framework Integrating WCET Analysis for Object-Oriented Real-Time 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

This paper proposes a novel gain time reclaiming framework integrating WCET analysis for object-oriented real-time systems in order to provide greater flexibility and without loss of the predicability and efficiency of the whole system. In this paper we present an approach which demonstrates how to improve the utilisation and overall performance of the whole system by reclaiming gain time at run-time. Our approach shows that integrating WCET with gain time reclaiming not only can provide a more flexible environment to develop object-oriented real-time applications, but it also does not necessarily result in unsafe or unpredictable timing analysis.

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

1. Introduction

There is a trend towards using object-oriented programming languages, such as Java and C++, to develop real-time applications. 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 bring about 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 WCET of each task is known prior to doing the schedulability analysis. Typically, the WCET analysis and schedulability analysis are carried out separately. On the one hand, sophisticated techniques for WCET analysis [6, 14, 13], for instance to model caching and pipelining, are used in order to achieve safe and tight WCET estimation. However, most WCET analysis approaches are only considered in relation to procedural programming languages. Some research groups have proposed various approaches [8, 15] to address these issues, but most approaches result in developing environments which are inflexible and very limited. On the other hand, in order to develop more flexible real-time systems, a number of research groups have proposed various flexible scheduling algorithms [5, 12], for instance priority server algorithms [5] and slack stealing algorithm [12]. In general, these flexible scheduling algorithms are mainly focused on the use of WCET to improve the performance of the aperiodic tasks at run-time. They have, however, paid insufficient attention 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.

In general, the spare capacity of the real-time system may be divided into three groups [7]: *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. The gain time is produced when the hard real-time tasks execute in less than their worst-case execution times. This may only be reclaimed at run-time since it depend on the actual executions of the tasks [7]. The spare time may be defined as a situation in which the 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 some research approaches [9, 1] have discussed how to reclaim the gain time. However, they have tended to focus on procedural programming languages, rather than on object-oriented programming languages.

In this paper we propose an approach which demonstrates how to improve the utilisation and performance of the whole system by reclaiming gain time at run-time. We use a gain time reclaiming mechanism to compensate for the tradeoff among the flexibility, efficiency and predictability. Our approach shows that integrating WCET analysis with gain time reclaiming not only may achieve high utilisation and high per-

*This work has been funded by the EPSRC under award number GR/M94113

formance of the whole real-time system, but also keep the flexibility of the object-oriented real-time applications. The major contributions of this paper are:

- presenting how to address the dynamic behaviour of object-oriented programming features with minimum annotations
- demonstrating how to reclaim the gain times of object-oriented real-time systems with the gain time reclaiming graphs
- balancing the flexibility and predicability of object-oriented real-time applications by integrating WCET analysis

The rest of the paper is organised as follows. Section 2 gives an overview of our previous work. Section 3 demonstrates how gain times can be reclaimed in object-oriented real-time systems. Finally, the conclusion and future work are presented in Section 4.

2. Previous Work

Our previous work, called Extended Real-Time Java (XRTJ)[11], extends the current Real-Time Java architecture [4] proposed by the Real-Time Java Expert Group. The XRTJ architecture has been developed with the whole software development process in mind: from the design phase to run-time phases. For example, using our approach, the system can be evaluated during the design, and the timing constraints of the application can be validated during run-time. We integrate our approach with the portable WCET analysis, proposed by Bernat et al. [3] and extended by Bate et al. [2], for the WCET estimation.

In our previous approach [11], we have introduced the *Extensible Annotations Class* (XAC) format, which stores extra information that cannot be expressed in the source code. The XAC format is an annotation structure that can be stored in files or as an additional code attribute in *Java Class Files* (JCF). We have also addressed dynamic dispatching issues in object-oriented real-time applications [10]. Here, minimum annotations are provided to ensure the predictability of dynamic binding methods and estimate safe and tight WCET for hard real-time applications. However, our previous work mainly focused on the analysis of the hard real-time object-oriented tasks.

3. Gain Time Reclaiming

In order to balance the tradeoff between the flexibility and efficiency of the real-time systems, gain time reclaiming needs to be applied. For the most part, the gain time reclaiming in object-oriented programming languages may be classified in three groups: *structural constraints reclaiming*, *functional constraints reclaiming*, and *object constraints reclaiming*. We use some WCET annotations, which are presented in our previous approaches [11, 10], in figures 1 and 2 . Further details of each reclaiming mechanism are discussed below.

//@ **GainTime**$(Units\ /path\ /mode\ /method)$	–A1
//@ **Dyn_GainTime**$(maxLoopcount, Scope\_Name)$	–A2
//@ **OO_GainTime**$(Object\_Name)$	–A3

Table 1. Gain Time Reclaiming Annotations

Note that the WCET annotations used in the following examples to discuss the gain time reclaiming mechanism can be added either manually by developers or automatically by modified compilers or tools.

3.1. Structural Constraints Reclaiming

From the point of view of the syntax of the programming languages, the real-time tasks allow construction with a number of basic blocks, conditional branches, and call procedures. These components of a real-time task, in general, may be represented by a *control flow graph* (CFG). It can be observed that the actual execution time of real-time tasks may vary, if the execution paths of the task or iteration times are varied at run-time. This section is mainly concerned with reclaiming gain times, which depend on the structural constraints of a specific real-time task.

```
1  ...
2  public check_data () {
3    int i , morecheck, wrongone;
4    i =0; morecheck=1; wrongone=−1;
5
6    //@ DefineScope(checkLoop)
7    while (morecheck) {
8      // Say WCET=20 cycles
9      if (data[ i ] < 0) {
10       //@ GainTime(100 cycles);
11       wrongone=i; morecheck=0;
12       //@ GainTime (Error_mode);
13     }
14     // Say WCET=120 cycles
15     else {
16       ...
17       if (++i >= DATASIZE)
18         morecheck=0;
19     }
20   }
21   //@ Dyn_GainTime(50, checkLoop);
22    ...
23
24   // Say WCET=10 cycles
25   if (wrongone >= 0) {
26     //@ Mode(Error_mode);
27      ...
28     return 0;
29   }
30   // Say WCET=50 cycles
31   else {
32     //@ Mode(Noml_mode);
33      ...
34     return 1;
35   }
36 }
37 ...
```

Figure 1. An example of gain time reclaiming [13]

Our approach is similar to Audsley et al.'s approach [1], which is proposed for procedural programming language. We have defined two annotations ($A1$ & $A2$), which are given in Table 1, to cope with structural constraints reclaiming. As shown in Figure 1, we can annotate the static gain time, such as precalculated unites or paths, with annotation $A1$, and the dynamic gain time, defined for unknown iteration times, with annotation $A2$. In Figure 1, the *if-then-else* basic block can reclaim 100 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. With respect to the dynamic gain time, we can simply add an annotation to a non-constant iteration loop, such as *for-loop* or *while-loop*, in order to reclaim gain times at run-time.

Essentially, the gain time can be reclaimed as soon as the

exact execution path of the task or iteration time are identified. Note that 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 addition code must be introduced by an annotation aware compiler to count the loops. It should also be noted 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.

3.2. Functional Constraints Reclaiming

This section is mainly concerned with reclaiming the gain times which suffer from functional constraints. This covers the issues that remain from the previous sections, which did not take into account the functional and data dependencies of the exclusive paths or modes of the real-time task.

Identifying the *exclusive paths* [13] or various *modes* [6] in order to calculate the WCET estimation of the real-time program is widely used in the WCET field. Based on design knowledge, the annotations of the exclusive paths or modes may be distinguished during the design phase. Using these annotations, the WCET estimation of each exclusive path or mode may be calculated. 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. As a result, the WCET estimation could be very pessimistic. In order to address this, we propose a gain time reclaiming framework which takes into account the functional constraints of the structure of the programs.

In our approach, we use the gain time annotation ($A1$), given in Table 1, to identify where the exclusive path or mode can be determined. As soon as the specific execution path or mode is determined or executed, the associated gain time of the executed path or mode can be reclaimed. Again using the previous example in Figure 1, the $A1$ annotation can be annotated at Line 12 to reclaim the functional associated gain time at run-time. It can be observed that using functional constraints reclaiming may reclaim the gain time earlier than the structural constraint reclaiming.

3.3. Object Constraints Reclaiming

So far, we have only discussed the gain time reclaiming which may apply to both procedural and object-oriented programming languages. 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 [10]. Our previous approach has shown that allowing the use of dynamic dispatching not only can provide a more flexible way to develop object-oriented hard real-time applications, but it also does not necessarily result in unpredictable timing analysis. Essentially, 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 too large or the WCET

```
/* ************************
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() {

    //@ OO_GainTime (aa);
    A aa= new A();
    //@ OO_GainTime (bb);
    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;
    }
    ...
    // type changing
    bb = cc;
    ...
    if (z == true) {
      ...
      aa.m1;
      ...
      aa.m1;
    } else {
      ...
      aa.m1;
    }
    ...
    bb.m1;
    ...
    bb.m1;
  }
}
```

Figure 2. An example of object constraints reclaiming

estimations for different classes are spread over a wide range of values. In order to compensate for the penalty of the flexibility of the object-oriented programming, gain time reclaiming is required.

Before discussing further details of the object constraint reclaiming, two novel terminologies are introduced below.

- An *Object Type Lifetime Graph* (OTLG) is a diagram which represents lifetimes of types of particular objects in a specific task. An OTLG is made of two types of component: node and edge. A *node* denotes a place where the type of the object is changed, whereas an *edge* illustrates the lifetime of a particular type of object between two nodes.

- An *Object Gain Time Reclaiming Graph* (OGTRG) is a diagram which illustrates places where the object constraint reclaiming may take place. An OGTRG also consists of two types of component: node and edge. A *node* denotes a place where the gain time can be reclaimed, whereas an *edge* illustrates that there is no gain time reclaiming taking place.

Essentially, the value of the dynamic dispatching gain time of each object can be calculated as follows: *//@GainTime()= //@maxWCET()-//@UseWCET()*. The annotations of the object gain time reclaiming may be generated by using design knowledge or by producing an OGTRG. In order to reduce the run-time overhead, annotation $A3$ may be applied to define which object's gain times are going to be reclaimed. The procedure of object gain time reclaiming is given as follows.

The *control flow graph* (CFG) can be produced from the source code (or Java class file) for each hard real-time task. Based on the CFG, the OTLG, which illustrates the lifetime

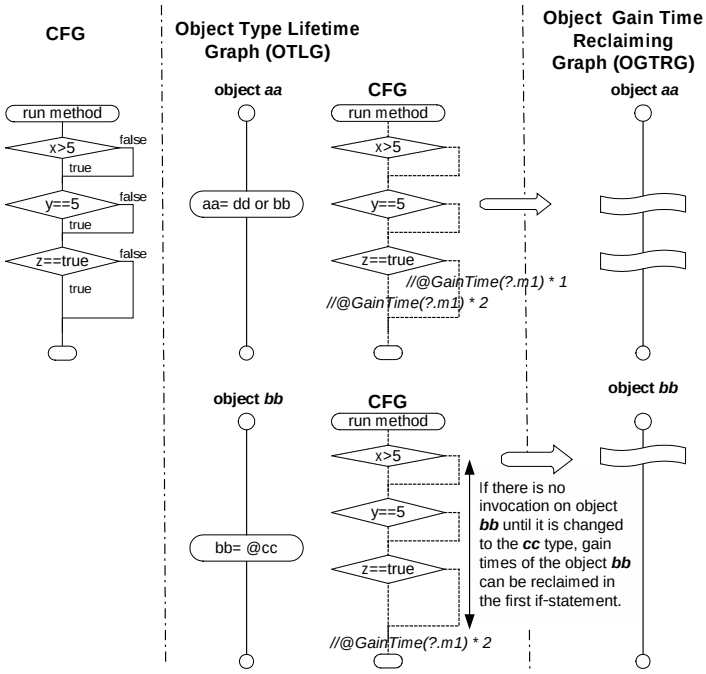

Figure 3. A diagram of producing OGTRG

of an object, can be produced. In the OTLG diagram, symbolic references may be applied to represent the relationship between the dynamic dispatching objects of the same class family during run-time. Using the CFG diagram and the OTLG diagram of each object, the exact places and amounts of gain time reclaiming can be identified. These gain time reclaiming places can be illustrated in an OGTRG for each object. Following this, the gain time reclaiming of all objects in the real-time task can be merged together and provided for the run-time environment (or Java virtual machine) to reclaim them. A diagram which illustrates the transformation from CFG to OGTRG is given in Figure 3.

Solving the symbolic expression of an associated class family can improve the reclaiming as early as possible. As shown in figures 2 and 3, the gain time of the object bb can be reclaimed as soon as the type of the object cc is determined.

4. Conclusion and Future Work

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

Here, we have mainly discussed the dynamic behaviour of object-oriented features, which is exclusively restricted to a consideration of the language syntax and semantic aspects. In order to cover as much dynamic behaviour of the object-oriented programming features as possible, our future work has to take into account: memory management, dynamic loading and extension, and remote method invocation (RMI) issues.

References

[1] N. C. Audsley, R. I. Davis, and A. Burns. Mechanisms for Enhancing the Flexibility and Utility of Hard Real-Time Systems. *In Proc. of the 15th IEEE Real-Time Systems symposium (RTSS)*, 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. *In 6th IEEE Real-Time Computing Systems and Applications (RTCSA2000)*, pages 39–48, December 2000.

[3] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. *In proc. 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. *In Proc. of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] F. Mueller. *Static Cache Simulation and its Applications*. Ph.d thesis, Department of Computer Science, Florida State University, July 1994.

[15] 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.