15th Triennial World Congress, Barcelona, Spain

JAVA FOR REAL-TIME PROCESS CONTROL SYSTEMS

Michael J. Baxter and Sian Hope

School of Informatics
University of Wales, Bangor,
United Kingdom
Email: mjbaxter@informatics.bangor.ac.uk

Abstract: This paper investigates the temporal behaviour of the Java run-time environment for real-time control system applications. A representative process controller is used as a case-study, and has been implemented in Java according to a number of proposed generic software architectures. These are assessed upon several general purpose run-time platforms, before extending the investigation to a claimed real-time Java environment. The investigation is then furthered, by compiling the Java into native code, in an attempt to improve execution time. The results prove surprising and indicate further directions in which to progress this technology. *Copyright © 2002 IFAC*

Keywords: *Process control; Distributed systems; Real-time systems.*

1. INTRODUCTION

This paper describes work that forms part of the PiCSI (Process Control Systems Integration) research programme that collaborates with companies involved in the process control industries, including: OAC, Eurotherm and Wind River Systems. It has been found that the process control industries are moving away from proprietary control solutions, and are interested in exploring fully decentralised distributed control systems (FDDCS), exploiting emerging SMART remote I/O technology. Industry estimates material cost savings to be in the range of 30-80% using this scheme. However, the scheme is not easily realised due to the increased design complexity, and requires the support of design tools and modern software technologies, possibly including Java (Bass, *et al*., 2000).

Java has often been mistaken as a language for web technologies only; of course it is much more. A general purpose programming language, object-oriented, heavily typed, distributed, robust, multithreaded, reusable code and with great strengths in portability (Horstmann, *et al*., 1997). Real-time systems engineers have regarded Java as an attractive language for all these features, but have been fundamentally unable to embrace the technology, as it cannot capture the temporal determinism required by these systems. A number of impacting issues have been raised in the real-time community in this regard, such as memory management, scheduling, inter-process communications and synchronisation. Java has been selected as the real-time target language for the PiCSI programme.

This paper will continue with a discussion of background material and industrial developments followed by an examination of the experimental approach, software architectures and the development environment used. Results are presented next and the paper will ends with conclusions and a discussion on future work.

2. BACKGROUND

The typical run-time environment for Java is unlike that for most other languages. To understand this, consider Fig. 1, it outlines the usual layers in the Java environment. The Java application sits at the top level of the architecture. Here, the application code does not usually run natively, it runs on a virtual machine, the Java virtual machine (JVM). The output of compiled Java source code are byte-codes which can be considered analogous to machine instructions. It is these byte-codes that are executed, or more accurately, interpreted by the JVM. Simply stated, a JVM can be seen as a software abstraction of a computer.

The JVM provides the portability of Java, as any hardware platform can execute Java providing it has a conforming JVM. The JVM includes mechanisms such memory management and scheduling. The JVM in turn calls on the services provided by the operating system, which sits upon the computing hardware.

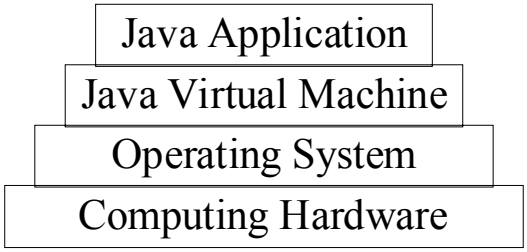

Fig. 1: Layers of a typical Java environment

Obviously, this is a more complex run-time architecture than that which is usually associated with a programming language like 'C'. Unfortunately, this also detracts from the execution performance and predictability of Java.

Within the execution environment of Java, there are undesirable activities. Threads have come under scrutiny, as their temporal requirements cannot be easily expressed and certainly not guaranteed (Miyoshi, *et al.*, 1997). More advanced features permit compilation during run-time, through the availability of just-in-time (JIT) compilers, such as those available from Symantec and Inprise. These compilers are automatically engaged during run-time, compiling code on-the-fly. Claiming, this provides speed-up of 10-20 times for some applications (Horstmann, *et al.*, 1997). However, this is undesirable for real-time systems, as it is difficult to predict when a JIT compilation will occur. There is also another technique, called ahead-of-time (AOT) compilation, which permits compilation to native code prior to run-time. However, it has been found that for some AOT compilers only the methods of an application are compiled, and the application is still run via the services of a JVM.

Interestingly, there is no reason why Java cannot be executed on conforming hardware directly, which simplifies the run-time architecture and improves the ability to perform real-time analysis. J. Kreuzinger *et al*, have made progress in this direction, with the development of the Komodo microcontroller, which executes a variant of standard Java natively (Kreuzinger, *et al.*, 1999). Other examples of Java hardware come from aJile Systems and Systronix who have developed single board Java computers based on the Rockwell Collins JEM2 core.

Industry speculates that Java is desirable for real-time systems and as a result of the shortcomings, a number of consortia have been assembled in an attempt to formulate specifications of a real-time Java. Encompassing many blue-chip companies in their membership, the two most significant groups are the Real-time for Java Experts Group (RTJEG) (www.rtj.com), and the other is the J-Consortium (www.j-consortium.com) (Bollela, *et al.*, 2000; The J Consortium, 1999; Baxter, *et al.*, 2000). Furthermore, there are a great variety of claimed real-time Java environments that have been released by a plethora of companies, including: Sun Microsystems, Inc. (java.sun.com); Newmonics, Inc. (www.newmonics.com); Insignia Solutions (www.insignia.com); Esmertec, Inc. (www.esmertec.com); aJile Systems (www.ajile.com) and Systronix, Inc (www.systronix.com).

3. THE INVESTIGATION

Considering, all the previous criticisms, it can be suggested that Java does not have a place in real-time systems. However, this paper makes a practical investigation through the simple benchmarking of a number of Java environments running a challenging process control case-study application. Initially, the authors propose several candidate software architectures suitable for representing control algorithms. The case-study is subsequently coded according to these architectures, and evaluated for execution cycle-time. The application is cyclic, where data is pumped into the inputs of the system, which percolates through the various software objects. The completion of a cycle occurs when all the data has been processed and data arrives at the output. It is this cycle-time that is of interest to this investigation. The implementation was prepared for timing analysis by developing a timing harness that would iterate the implementation many times.

4. CASE-STUDY APPLICATION

The case-study has been drawn from the process control industries, and is illustrated in Fig. 2. It is a component of a controller for an extruder process, which is known to be a challenging control problem. The application was specified and simulated using the Mathworks Simulink environment (Mathworks, 2000), at the Control Systems Centre (CSC) at UMIST, Manchester. A portion of the complete application was then hand coded in Java using the proposed software architectures. This controller is cyclic, and has a time period requirement of 34mS. Failure for the application to be computed within this time will affect the dynamics of the system, which is considered unacceptable. The first objective of this investigation is to assess whether the algorithm can be computed within this stringent time constraint.

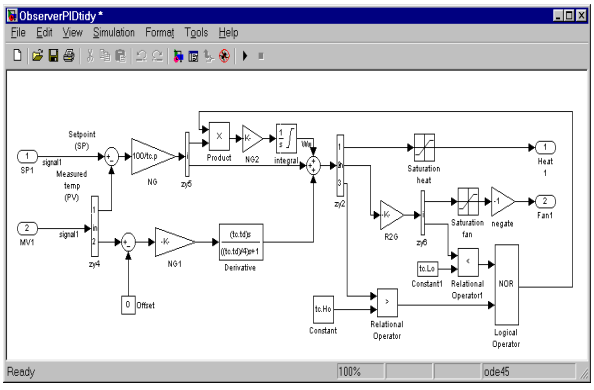

Fig. 2 Simulink view of case-study application

5. SOFTWARE ARCHITECTURES

The researchers propose a number of generic software architectures that could be used to represent a typical controller; they are based on the results of a previous benchmarking experiment (Baxter, *et al.*, 2000). These were:

a) An elegant and multi-threaded architecture, based on the communicating sequential processes (CSP) parallel processing paradigm (Hoare, 1985; Kreuzinger *et al.*, 1999).

b) A simple serialised architecture, proposed for efficient run-time performance.

c) A multithreaded architecture with increased granularity and fewer threads, to overcome the performance inefficiencies of (a) and the lack of object-orientation in (b).

d) A multi-threaded architecture, that exploits method call communications, based on (a) but with potential execution performance benefits.

5.1. Multi-Threaded Architecture

This novel, yet elegant, software architecture is based on CSP, proposed by Hoare (Hoare, 1985). Consider the case-study in Fig. 2; it illustrates the Simulink block diagram for an advanced PID controller. It was noted that the controller data-flow diagram is constructed from transfer-function blocks and interconnecting signals. As can be seen, the diagram is essentially a data-flow diagram. This structure can be elegantly represented by the CSP paradigm, which is rooted in mathematical formalism and is analysable.

To understand the transition to code consider Fig. 3, it illustrates a simple controller in Simulink. The resulting multithreaded architecture can be seen in Fig. 4, which is structurally similar. Here, each block of the Simulink diagram is translated into a Java thread. The corresponding interconnecting signals in Simulink are constructed with streams.

An elegance of this multi-threaded approach is that it takes advantage of the regularity of control system diagrams. Control algorithms are typically constructed from a small library of block types, for example: transfer functions, adders and multipliers. A library of Java classes, each representing a block type has been built. From here, the implementation of a controller only requires code to instantiate the relevant blocks and signals with an appropriate harness that 'wires' the components together, see Listings 1 for the generic pseudocode. Initially, the constructor method instantiates all the signals and blocks, and connects the streams. The blocks are subsequently named, for debugging purposes. Finally, the threads are set to run. The run method forms the body of the code, and is an endless loop that wraps around the system input and output.

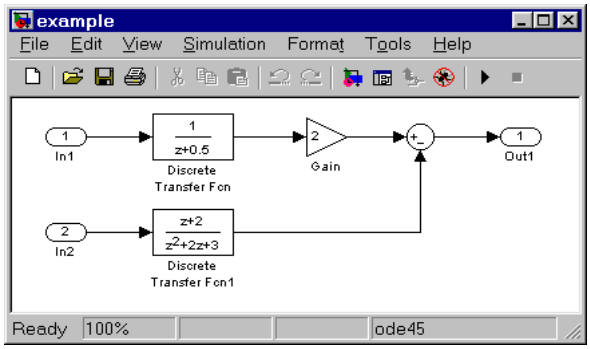

Fig. 3 An example controller in Simulink

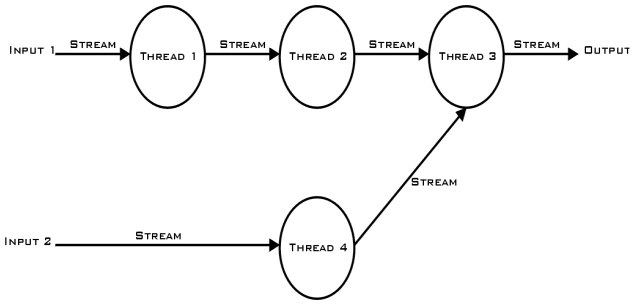

Fig. 4 Example multi-threaded software architecture

```
class CONTROLLER extends Thread
{
  private DECLARE SYSTEM INPUTS
  private DECLARE SYSTEM OUTPUTS

  public CONTROLLER ( I/O PARAMETERS )
  {

    INSTANTIATE INPUT SIGNALS
    INSTANTIATE OUTPUT SIGNALS

    Line DECLARE SIGNALS = new Line();

    TEMPLATE BLOCK = new TEMPLATE
    (
      BLOCK INSTANCE PARAMETER VALUES,
      BLOCK_INPUTS,
      BLOCK_OUTPUTS
    );

    SET HIERARCHIAL NAMES OF ALL BLOCKS
    START ALL CREATED THREADS
  }

  public void run()
  {
    while( TRUE )
    {
      READ SYSTEM INPUT
      SEND INPUTS TO CONTROLLER
      READ OUTPUTS FROM CONTROLLER
      WRITE SYSTEM OUTPUT
    }
  }
}
```

Listings 1: Multi-threaded harness pseudocode

An advantage of this simple architecture is that the precedence of blocks in the data-flow diagram does not need to be considered in the implementation. Threads will only be run if they have input data, if they don't, they will block and are not scheduled. In addition, to minimise the effects of garbage

collection the application has a static memory requirement. After initialisation of the application at run-time, no further memory space is requested.

```
class BLOCKNAME extends Thread
{
  private double I/O VARIABLE DECLARATIONS
  private double OTHER VARIABLE DECLARATIONS
  private PIPE VARIABLE DECLARATIONS

  public void BLOCKNAME( PARAMETERS )
  {
    INITIALISE I/O VARIABLE DECLARATIONS
    INITIALISE PIPE VARIABLE DECLARATIONS
  }

  public run()
  {
    while( TRUE )
    {
      READ INPUTS
      EXECUTE ALGORITHM
      WRITE OUTPUTS
    }
  }
}
```

Listings 2: Generic class pseudocode

The classes within the library are all constructed around a common structure, see Listings 2. The class contains a constructor and a run method. The constructor initialises parameters for this instance of the block type, and is also responsible for attaching the ends of the streams that will be used for inter-thread communications. The main body of the code appears in the run method, this is the code that is executed once the thread has been prepared for execution. Here, a non-terminating loop wraps around the mathematical equations representing the block, with appropriate input and output operations. There are a number of other software architectures that could be used here, but this was initially chosen for simplicity.

It is important to note that this simple relationship between the Simulink diagram and Java code lends itself very well to automatic code-generation. A code-generator could draw on this class library of controller components, and by building the communication harness based on the generic pseudocode described, a complete controller is easily produced. The case-study here was handcoded, which when considering more complex examples was not considered a scalable approach. For automatic code-generation, this increased complexity is not expected to be an issue.

5.2. Serialised Architecture

Multi-threaded applications incur run-time overheads due to scheduling and context switching overheads (Baxter, et al., 2000). Although, here, the virtual machine will be kept busy whilst there are runnable tasks. To investigate the burden of multi-threading, an alternative software architecture was proposed, based on a serialised structure. Consider Listings 3, it represents the pseudocode for a generic serialised architecture. Here, a single thread represents the complete controller. The blocks are implemented as in-lined code and the signals as local variables. An endless loop, wraps around the controller maths, firstly external inputs are read. Subsequently, the individual blocks are computed with the outputs being written at the end. The result is an architecture that does not introduce the threading overheads previously described. However, this architecture does require analysis of block precedence, the developer must order the code to match the diagram. In addition, it does not take advantage of the power of object orientation, as only one object is in the system, the complete controller. As with the multi-threaded architecture, the application has a static memory requirement.

```
public class CONTROLLER_NAME
{
  public void CONTROLLER_NAME()
  {
    double COMM VARIABLE DECLARATIONS
    double BLOCK PARAM VARIABLE DECLARATIONS
    double SYSTEM VARIABLE DECLARATIONS

    while( TRUE )
    {
      READ INPUTS
      PROCESS BLOCK 1
      PROCESS BLOCK 2
          . . .
          . . .
      PROCESS BLOCK N
      WRITE OUTPUTS
    }
  }
}
```

Listings 3: Serialised architecture pseudocode

5.3. Large-grained Multi-threaded Architecture

The multi-threaded approach can be criticised for the overheads that are introduced due to thread management, such as, context switching and scheduling. In addition, the code representing the simulink blocks does not have a high computational demand, but requires significant communication across pipes. This indicates fine-grained tasks, which are known to be inefficient. However, the serialised approach eliminated these issues, but does not exploit object-orientation and results in unmanageable code size for real-world controllers.

These arguments led the authors to propose a multi-threaded architecture with increased task granularity. This is performed by creating small groups of neighbouring simulink blocks to form larger-grained blocks, essentially a process of serialisation. Subsequently, these are implemented as threads in accordance with the multi-thread architecture.

5.4. Multi-threaded architecture with method call communications

Existing benchmarks (Baxter et al, 2000) indicate that a single communication over a pipe incurs a fixed overheads for messages sizes upto 512bytes, for the virtual machines investigated. This communication overhead becomes comparatively significant for the small message sizes used here. This led the authors to consider an architecture that

uses method calls for communications, believed to have comparative performance benefits for small message sizes. Here, the multi-threaded architecture is retained, but streams are substituted for method calls.

6. EXPERIMENTAL ENVIRONMENT

The Borland JBuilder3 integrated development environment was used for the initial application development, and then compiled to byte-codes by the Sun Microsystems Java compiler from version 1.3 of the standard development kit. At this stage, the byte-code application can be run on the chosen JVMs. Two different JVMs are used, the standard Sun JVM and another commercial offering, which will be designated as Brand X. However, the Brand X toolset suite also has provisions for compiling byte-codes into native code. This requires the use of the accompanying tool and a 'C' compiler, from which a standalone executable is built. Finally, the GNU 'gcj' is also used, that compiles Java source code directly to native code (gcc.gnu.org). The underlying operating systems are Windows 98 and Redhat Linux 7.x. The list below summarises the experimental configurations, which were used in the investigation:

- Sun JVM on Windows 98
- Sun JVM on Linux
- Brand X JVM on Linux
- Brand X compiled native executable on Linux
- GNU 'gcj' compiled native executable on Linux

The hardware used for all these investigations was an Intel Pentium II 400MHz on an Intel SE440BX-2 motherboard with 128MByte of system memory.

7. RESULTS

Table 1 outlines the cycle-time results for a selection of the specified configurations. For the application in hand, the Java multi-threaded byte-code implementation upon Windows 98 is not performing well, when considering the requirements of the algorithm. In fact, it takes nearly 200 times longer than the required 34mS cycle-time. However, transporting the application to the Linux OS provides a modest reduction in execution time. Although the Brand X JVM is claimed to provide improved execution and context switching performance over the Sun implementation, only a small cycle-time advancement was shown. Subsequently, the application was compiled into native-code utilising the Brand X tools. The authors had a confident expectation that the requirements would be satisfied using this approach; there was a huge disappointment when only a modest improvement was demonstrated.

However, the serialised architecture resulted in massive performance benefits, when compared to the multi-threaded architecture. Although the timing requirement was not met, with this simple software architecture the result is very close. The improvements are in excess of 100 fold, when compared to the multi-threaded variant, and are clearly down to the reduction of scheduling overheads and inter-thread communications.

Table 1: Application cycle-time results

Software Architecture	App' Code Format	Virtual Machine	OS	Cycle-time (S)
Multi-threaded	Byte-code	Sun	Win98	6.59
Multi-threaded	Byte-code	Sun	Linux	5.11
Multi-threaded	Byte-code	Brand X	Linux	5.01
Multi-threaded	Native-code	Brand X	Linux	4.95
Multi-threaded time-slice tuned	Native-code	Brand X	Linux	1.61
Serialised	Byte-code	Sun	Win98	0.045
Serialised	Byte-code	Brand X	Linux	0.043
Serialised	Native-code	GNU	Linux	0.040

Additionally, the authors also tried tuning the scheduler time-slice for the Brand X native-code variant, performed on the command-line. Improvements were in the order of 3 fold. The default was 25mS, and the best setting tried was 8mS.

Finally, the GNU compiler was applied to the serialised version. Again, a small improvement was demonstrated over the other serialised implementations, but still short of the temporal demands.

8. CONCLUSION

Despite the plethora of configurations trialed, none of implementations met the 34mS cycle-time requirement. But before concluding that Java has no place in real-time process control systems, reconsider the table of results. Obviously, the choice of JVM and OS has not had significant impact. Neither has the compilation to native-code, which surprised the authors, and questions were raised on why there was little improvement. On further investigation it was found that full AOT was not occurring, only the compilation of methods, and the application was still being run upon the services of the JVM. Here, the term AOT is considered misleading.

The most significant improvement was a result of the software architecture. It was expected that the multi-threaded architecture would incur run-time overheads due to scheduling and context switching. However, the results of the serialised code imply that these overheads are massive, when compared to the time spent executing the control algorithm itself. Tuning

the scheduler time-slice also released some benefits. This reduction was again surprising; possibly indicating the threads may not be yielding to other runnable threads when blocked on communications. However, this must be confirmed through further consultation, experimentation and extended to the other configurations.

Clearly, the run-time architecture of the Java platform is far more complex than running compiled 'C' directly on a microprocessor. This paper has proposed a number of software architectures and performed a pragmatic trial, with unexpected results. In light of these, it is not obvious what constitutes a good software architecture without a practical trial, and efficient use of the multi-threading features of Java is challenging, despite an elegant software representation of the controller.

9. FUTURE WORK

Clearly, the interactions of software components and run-time mechanisms are complex within a Java environment. The software architectures investigated in this paper were partially proposed on the basis of some simple benchmarking of execution and communication performance benchmarks. There may be merit in extending these focused benchmarks, to better characterise the alternative platforms. With this information the architectures could be revised and further execution performance improvements may be released. In addition, there are two further architectures that will be trialed, which address some of the shortcomings of the multi-threaded and serialised approaches.

The investigation has focused on one case-study only. It is considered a good representation of a challenging control algorithm, but has limited investigative scope. This will be furthered by identifying other general algorithms, in order to demonstrate the approaches outlined in this paper.

As described, full AOT compilation did not occur with the Brand X tools. Others are available, such as the GNU Java compiler and there maybe merit in trialing these also. Full compilation is expected to release massive improvements, but this must be confirmed.

Obviously, the run-time environments in this study are not hard real-time. The virtual machines make use of some of the underlying services of the operating system, such as the scheduler. However, there maybe potential rewards through the exploitation of the awarding winning Wind River Systems' VxWorks real-time operating system, in conjunction with a compatible virtual machine, such as a recent Sun Microsystems release.

For some time, Java processing in hardware has been considered an attractive proposition. Recent developments from aJile Systems are exciting, and there may be some merit in investigating the suitability for real-time systems. Initially, applying the previous benchmarks and case-study application may demonstrate that a hardware approach is more viable. Moreover, for hard real-time systems this approach maybe far simpler to analyse. For example, the determination of worst-case execution times (WCETs) for software components is not straight forward in the usual multi-tier Java architecture. This would be greatly simplified when using Java hardware, as the core is micro-coded and byte-codes would be expected to have a fixed worst-case execution time. For example, traditional instruction counting techniques could be applied to determine WCETs.

ACKNOWLEDGEMENTS

This paper acknowledges the support of the EPSRC, grant number GR/M55282. The paper also acknowledges the technical contributions made by the Process Control Systems Integration Industrial Consortia, of which the authors are members.

REFERENCES

Bakkers, A., Hilderink, G. and Broenink, J., (1999). 'A Distributed Real-time Java Systems Based on CSP', *Proceedings of WoTUG 22: Architectures, Languages and Techniques for Concurrent Computing,* pp.229-241

Bass, J.M., Schooling, S. and Turnbull, G. (2000). 'Process Control Systems Integration', *Proceedings of IFAC CACSD 2000*.

Baxter, M.J. and Bass, J.M., (2000). 'Achieving Hard Real-time Java', *Proceedings of the 25th IFAC Workshop on Real-time Programming,* pp.161-166

Baxter, M.J., Hope, S., and Bass, J.M., (2001). 'Java for Distributed Real-time Systems: Practical or Intractable?',…….

Bollella, G., Gosling, J., Brosgol, B.M., Dibble, P., Furr, S., Hardin, D. and Turnbull, M. (2000), *The Real-Time Specification for Java,* Addison-Wesley, ISBN 0-201-70323-8.

Hoare, C.A.R., (1985). *'Communicating Sequential Processes',* Prentice Hall International Series in Computer Science.

Horstmann, C.S. and Cornell, G., (1997). *'Core Java',* Vol 1 - Fundamentals.

Kreuzinger, J., Marston, R., Ungerer, Th., Brinkschulte, U. and Krakowski, C., (1999). 'The Komodo Project: Thread-based Event Handling Supported by a Multithreaded Java Microcontroller', *Proceedings of the 25th EUROMICRO Conference,* Vol2, pp.122-128.

Miyoshi, A., Kitayama, T. and Tokuda, H., (1997). 'Implementation and Evaluation of Real-time Java Threads', *Proceedings of the 18th IEEE Real-time Systems Symposium,* Vol 31, pp.166-175.

The J Consortium, (1999). *'Real-time Core Extensions for the Java Platform',* Draft, Version 1.0.2

The Mathworks, Inc. (2000) Using MATLAB