Developing Predictable Real-Time Embedded Systems using AnvilJ

Ian Gray
Real-Time Systems Group
University of York, York, U.K.
ian.gray@cs.york.ac.uk

Neil C. Audsley
Real-Time Systems Group
University of York, York, U.K.
neil.audsley@cs.york.ac.uk

***Abstract*—This paper proposes AnvilJ, a novel technology developed to assist the development of software for predictable, embedded applications. In particular, the work focuses on the complexities of programming for heterogeneous embedded systems in an industrial context, in which the need for predictability is an important requirement. AnvilJ converts architecturally-neutral Java code into a set of target-specific programs, automatically distributing the input software over the heterogeneous target architecture whilst ensuring preservation of predictability. During translation it generates a low- to zero-overhead runtime that is tailored to the specific combination of input application and target system, thereby ensuring maximum efficiency. AnvilJ uses a technique called Compile-Time Virtualisation that allows it to work with existing compilers and removes the need for language extensions which can hinder certification efforts.**

***Keywords*-Java, AnvilJ, Compile-Time Virtualisation, Embedded Systems, Real-Time Systems, MADES.**

I. Introduction

Modern embedded systems present unique challenges for software development. The programming model of mainstream languages (such as C or Java) assumes a homogeneous architecture with a uniform, shared memory space. This model is incompatible with the trend in embedded systems to use application-specific, heterogeneous architectures [16] that contain multiple processing units of differing capabilities (processors, DSPs, FPGAs, etc.) and multiple non-uniform memory blocks of differing sizes, speeds, and technologies (such as transactional or scratchpad memories [4]). This results in a mismatch between the programmer's conceptual model and the underlying implementation.

The second challenge is that embedded systems are frequently deployed in safety-critical or high-integrity domains. This requires the systems to be predictable in terms of execution time and resource usage for correctness and schedulability analysis. The growing size and complexity of embedded systems means that achieving state coverage (and therefore verification) using simulation- or measurement-based approaches alone is difficult.

Languages like C, C++, or Java have started to allow the developer limited capability to reason about the target architecture and the way in which their software should be mapped to it. For example, Java's Real-Time Specification [10] (RTSJ) and POSIX allow basic mapping of threads to processors and to model physical memory, but largely homogeneous architectures are still assumed. Developers must rely on language extensions or extra-linguistic techniques (e.g. custom tools and linker scripts) to fully exploit complex hardware.

This paper argues there is a need for a form of lightweight, low overhead, virtualisation which can hide the complexities of developing software for non-standard embedded architectures, whilst still maintaining predictability and industrial applicability. Existing systems tend to either fail to effectively exploit changing underlying architectures, introduce too much overhead for resource-constrained systems, or introduce too much unpredictability to the final implementation.

This paper introduces AnvilJ[1], an approach for the predictable Compile-Time Virtualisation (CTV) [11] of heterogeneous architectures. AnvilJ provides predictable virtualisation of complex architectures using Java as its input programming language. AnvilJ is designed to preserve the predictability of the input system, to not introduce non-determinism or analytical complexity, and to produce a specially-tailored, minimal-overhead implementation. The same architecturally-neutral input code can be distributed over different target architectures simply by updating the system model. Verification is aided by CTV's aggressive use of static rather than dynamic behaviour, reuse of legacy code and existing tools, and the very small size of its generated libraries and runtime.

Section II discusses previous work and introduces Compile-Time Virtualisation. Section III describes AnvilJ and its system model, and then section IV describes the way it is implemented. Section V evaluates the overheads imposed by AnvilJ, and finally section VI presents conclusions.

II. Background

Existing attempts to target complex systems frequently involve creating new programming languages with extended programming models that specifically address a given problem. For example, languages for concurrency [7], non-uniform memory [9] or data streaming [24]. The problem with such an approach is that it is not amenable to the domain of high-integrity industrial applications. In industry, legacy code is important due to the cost of redevelopment and recertification.

[1]AnvilJ was developed as part of MADES - EU Framework 7 STREP project FP7-ICT-2007 248864 (see section V).

Also, switching to new tooling (compilers, linkers, etc.) is undesirable because compilers only attain "trusted" status after many years of use in the field. Companies are forced to either use trusted compilers, or to certify the output object code rather than the input source code, increasing certification effort.

Another common approach is to introduce a middleware or OS layer to mask shortcomings of the input language without a new compiler. However, such solutions [5], [20] introduce a large overhead cost, both in terms of static codebase size (which is a problem for certification) and runtime memory or processor use. Most systems also require the software to be manually partitioned into multiple input applications, rather than the more flexible single-program model in which the software is all one application and software elements (i.e. tasks or threads) are mapped over the architecture.

In general, middleware systems are still bound by the unpredictable programming model of their host languages. By design they hide low-level hardware details, which can lead to unpredictable timing behaviour, and must support undesirable features inherited from their source language (such as C's assumption of a single global address space). These factors can make such systems unsuitable for embedded real-time systems.

Compile-Time Virtualisation (CTV) is a technique developed to allow the exploitation of complex architectures from existing programming languages without language extensions or custom compilers. CTV replaces the inflexible layers of virtualisation and abstraction present in normal software development with a single virtualisation layer across the entire architecture, termed the *Virtual Platform* (VP). The VP is a compile-time concept, sitting between the input software and the compiler and is implemented through a combination of source-to-source translation and code generation. The VP presents an idealised view of the underlying hardware and target compiler, making it appear to the programmer that the source language and toolchain can target the given embedded architecture, where in reality it may not.

For example, C and C++ assume a single logical address space, but this is not true for many systems. The VP will expose a single address space and the programmer may code as if it is the case. At compile-time, the VP translates the input code into refactored code which uses the true memory architecture and takes care of the low-level, hardware-specific details.

CTV differs from existing virtualisation and translation-based approaches in the way that the VP is realised. Its existence at compile-time (rather than runtime) results in a very low-overhead system as required runtime support is generated each compile for exactly the input code and architecture of the system. CTV's novelty is discussed in detail in existing work, but can be summarised as follows:

- CTV operates on existing unmodified input code with standard compilers using compile-time code refactoring by providing a virtualisation layer that allows the programming model of the input language (normally homogeneous SMP or similar) to target heterogenous, multicore, embedded systems with non-uniform memory and arbitrary communications topologies.
- Virtualisation mappings (threads $\rightarrow$ CPUs, data $\rightarrow$ memory) are provided by the programmer, unlike in most runtime systems.
- In contrast to most runtime systems, CTV's compile-time system model assumes the communications and partitioning of the system is static and any dynamism must be explicitly enumerated.
- Runtime systems also tend to allow fully dynamic behaviour of the input code (thread migration, dynamic shared memory etc.). CTV's system model places well-defined restrictions on this model.
- Because of these restrictions, refactoring is predictable and traceable and the required runtime support is very small and timing predictable. Complex behaviour is only provided if explicitly required by the program.
- Also, they allow the runtime support to be built minimally for each program and architecture combination. Unused features are optimised away, minimising overheads.
- Porting programs to different architectures requires no code editing, only new virtualisation mappings.

CTV has already been used to target C code to heterogeneous FPGA-based architectures with hardware accelerators [11], and to integrate with MCAPI and MRAPI – industry standard APIs for embedded programming [12]. This work presents an implementation of CTV that is based on targeting Java (and its real-time subsets) at embedded architectures.

III. System Model

The abstraction layers and models used in most programming languages were developed to target fixed, uniform architectures. As general-purpose architectures evolved they introduced hardware features (e.g. Memory Management Units, Hyperthreading [16]) to hide their underlying complexity and emulate this simple model. However, these are frequently absent from embedded hardware. Also, as hardware continues to diverge from the basic model it is becoming impossible to hide the differences in an efficient and predictable way without assistance from the programmer.

As previously discussed, a system which supports legacy code and standard languages is desirable. The system must therefore introduce missing concepts to the programming model which allow the designer to map software elements to architectural elements and limit the implications of assumptions made by the programming language. For Java, the assumption of uniform shared memory makes most multi-core JVMs (e.g. Terracotta [21] or Kaffemik [2]) provide distributed shared memory over a uniform TCP/IP-based network. This is acceptable for large-scale systems, but heavyweight for embedded systems.

AnvilJ is an implementation of CTV that allows platform-independent Java (and its subsets) to effectively exploit complex, heterogeneous, embedded architectures with non-uniform memory architectures. This section will detail the system model of all CTV systems in section III-A, and then show

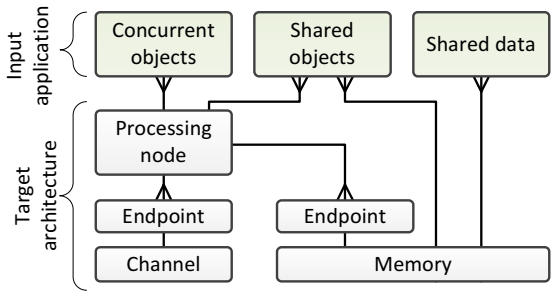

Fig. 1. The CTV system model

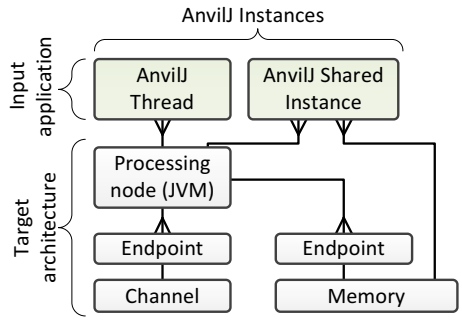

Fig. 2. The AnvilJ system model

how this model is refined for AnvilJ in section III-B. The assumptions made by AnvilJ for developing predictable systems are detailed in section III-C.

A. CTV System Model

CTV defines a language-neutral system model which represents a single input application and a deployment of that software over a target architecture. The single-program input allows great flexibility because the mappings of software to hardware can be adjusted to explore the design space without recoding the input program. Greater detail on this model is available in existing CTV work and is illustrated in figure 1.

The input to a CTV system is a single application containing:

- **Concurrent objects (CO):** Active elements with a thread of execution and local memory. These exist for the duration of the system and may create other language elements (tasks, data items, etc.) in the same computation node (see below).
- **Shared objects (SO):** Passive elements that expose a set of shared procedures that take arguments and return values. Invoked by the COs of the system. May contain state, stored in its memory space (see below). Implement a synchronization lock that can optionally be used to make its shared procedures mutually-exclusive.
- **Shared data:** Elements which merely contains state. Provide no guarantees about access control or coherence.

A CO may communicate with any other COs or SOs, however the elements it has created may not communicate with the created elements of other COs or SO. This restriction allows the communication topology of the system to be determined at compile-time and the required runtime support to be reduced, as discussed later. This approach is particularly suited to embedded development because it mirrors many of the restrictions enforced by high-integrity and certification-focussed language subsets (such as the Ravenscar subsets of Ada [6] and Java [14] or the MISRA-C coding guidelines [23]).

The COs in a CTV system are each mapped to exactly one **processing node** in the target architecture. Nodes are hardware elements that provide computation. They communicate with other nodes using **channels** which are shared communication resources that can be used to transfer messages. Channels have at least one **endpoint**, which models the connection between a node and a channel. Shared objects and data are mapped to **memory spaces** which provide logically-contiguous data storage with similar access patterns. A node may be connected to arbitrary numbers of endpoints.

This model is compile-time static - the number of COs, SOs, etc. does not change during runtime. This is in contrast to systems like CORBA which adopt a "dynamic-default" approach in which runtime behaviour is limited only by the supported language features. Such systems support a rich runtime model but the resulting system can be heavyweight as they are forced to support features such as system-wide cache coherency, thread creation and migration or dynamic message routing, even if not required by the actual application or architecture. The approach of CTV is "static-default" in which the part of the application modelled is static, resulting in a restricted programming model that promises less, but the amount of statically-available mapping information means that for most nodes of the system the required runtime can be significantly reduced.

B. AnvilJ System Model

AnvilJ is an implementation of CTV for the Java programming language and its related subsets aimed at ensuring system predictability, such as the RTSJ. Accordingly, it presents a system model (shown in figure 2) consistent with the CTV model from section III-A but with some refinements and clarifications. The input to AnvilJ is a single Java application modelled as containing two sets:

- **AnvilJ Thread:** (CTV Concurrent objects) A `static final` instance or descendant of `java.lang.Thread`.
- **AnvilJ Shared Instance:** (CTV Shared objects) A `static final` instance of any other class.

In Java, all data must be contained within an object instance (or static in a class) so CTV shared data items are not needed in the AnvilJ model. Collectively, AnvilJ Threads and Shared Instances are described using the umbrella term *AnvilJ Instances*. It is not strictly necessary to differentiate between Threads and Shared Instances as merely *instances* would suffice. Whilst this would better reflect the object-oriented nature of Java, the differentiation is used so that initial analysis and verification of mappings can be performed

without knowledge of the source program. It is clear from the mappings which instances are active and which are passive.

The move to support Java necessitated one significant change in the CTV system model. In CTV the architecture is modelled in terms of processors, in AnvilJ it is in terms of **processing nodes**. A processing node models a Real-Time Java Virtual Machine (JVM) [19] in the final system (or a standard JVM with accordingly reduced predictability). The Java specification does not define whether a multicore system should contain a single JVM for the entire system [2], [21] or one per core. Therefore to support this, AnvilJ needs to model the JVMs, rather than just the processors themselves. The JVMs need not have similar performance characteristics or features. As with CTV, every AnvilJ Instance is mapped to exactly one node.

Nodes communicate using **channels**, which are the communication primitives of the target architecture. AnvilJ statically routes messages across the nodes of the system to present the totally-connected communications assumed by Java. The designer provides drivers for the channels of the system (see section IV-D). **Memories** and **endpoints** are as in the existing CTV model. Every AnvilJ Shared Instance must be mapped to either exactly one node (on the heap of the JVM), or exactly one memory where it will be available to all nodes connected to that memory.

Not all instances of `java.lang.Thread` need to be modelled as an AnvilJ Instance. Equally, not all shared object instances need to be modelled at all. Enough should be modelled to fulfill the constraint (from the CTV model) that program instances created by an AnvilJ Thread t only communicate with other instances created by t, or AnvilJ Instances. This is known as the *'no dynamic shared data' constraint* and is discussed in detail in section III-E.

C. Assumptions

AnvilJ places some limits on the dynamic behaviour of the input software and the way in which it is mapped to the target architecture, as follows:

- The number of AnvilJ Instances is compile-time static.
- All statements of the input code which access an AnvilJ Instance can be determined offline.
- The 'no dynamic shared data' constraint (see section III-B).

These statements originate from CTV, and are translated into Java-specific terms in section IV-A. In practice, these restrictions are used because they are a natural way to develop embedded software when using the single-program model (one program to define the whole system, rather than the more common one-program-per-processor).

When developing AnvilJ, an alterative to these restrictions was considered. An 'elaboration phase' which executes on start up to initialise the application structure by instantiating threads and data structures can allow for predictable runtime behaviour (after the elaboration phase). This may be better for general-purpose frameworks because it limits the input software less. However embedded developers are used to such restrictions through coding guidelines and the use of static analysis tools. Requiring the overall structure to be compile-time static greatly simplified the implementation of the AnvilJ refactoring tools, and enhanced the potential for minimisation of runtime support.

AnvilJ does not require the use of the RTSJ because it aims to be applicable to all embedded systems. This is why the initial implementation uses standard Java (see section IV). If AnvilJ is to be used to make predictable systems the following additional assumptions are made:

- The computation model is that of coordinating threads, sharing data using synchronized shared memory.
- Each thread is assigned a fixed priority level - the executing thread is always the thread with the highest priority which is not otherwise blocked.
- Priority inversion in the final system can be prevented, or predicted and bounded.
- Threads contain code with bounded execution times.
- Blocking throughout the system (such as when accessing synchronised methods) is bounded.

These assumptions are derived from the RTSJ and Ravenscar Java. Note also the chosen hardware architecture should contain processors, memories and channels which allow bounded access times to be obtained. This paper will argue that the AnvilJ transformation supports *preservation of predictability*. This means that if the input system conforms to AnvilJ's system model then the output system will remain as predictable. [2]

D. Running Example

The code skeleton example in figure 3 shows the classic producer-consumer design pattern expressed in standard Java with a producer thread and two worker threads. This example will be revisited throughout the paper to demonstrate the way in which AnvilJ's refactoring operates, and that it preserves the predictability of the input program. The target architecture will be a dual processor node system with no shared memory, and a single communications channel between the two nodes. The main thread, `w1`, and `wq` are assigned to one node, `w2` to the other.

E. Assigning AnvilJ Instances

The 'no dynamic shared data' constraint (section III-B) allows the AnvilJ runtime to be drastically reduced from that of a full distributed JVM because it relaxes restrictions imposed by the Java programming model:

- It is not necessary to maintain expensive cache coherency across all nodes of the system.
- Communications are defined offline and are static, therefore only offline routing is required.
- The runtime behaviour of the system can only communicate between nodes at well-defined points, negating the

[2]The system will require feasibility analysis to determine response times etc. but AnvilJ does not introduce unpredictability.

```
public static final int WORKCOUNT = 100;

static class WorkQueue {
 synchronized Work getWork() {...};
 synchronized void putWork(Work w) {...};}

static class Worker extends Thread {
 public void run() {
  for(int i = 0; i < WORKCOUNT / 2; i++) {
   Work w = wq.getWork();
   ...}}}

final static Worker w1 = new Worker();
final static Worker w2 = new Worker();
final static WorkQueue wq = new WorkQueue();

public static void main(String[] args) {
 w1.start();
 w2.start();
 for(int i = 0; i < WORKCOUNT; i++) {
  ...
  Work w = createWork();
  wq.putWork(w);}}
```

Fig. 3. Example code skeleton, to be revisited throughout this paper

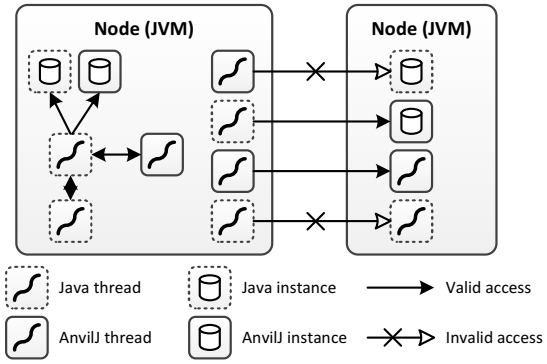

Fig. 4. Valid and invalid accesses in an AnvilJ program

requirement for new object instances to be propagated throughout the system.

This constraint represents the greatest challenge observed in moving Java into the CTV system model. Java's heavy use of object allocation makes this hard to analyse at compile-time. If the programmer does not designate enough instances as AnvilJ Instances and the constraint is not met, then the output programs may not have the same functional behaviour as the input program. Unfortunately, it is not possible in the general case to analyse the input program to ensure compliance (although AnvilJ's refactoring engine does check many analysable cases). We approach this problem by defining precisely under which circumstances incorrect behaviour will be observed. First, all behaviour that only affects a single node will execute correctly. Incorrect behaviour can only arise when data is shared across nodes, as visualised in figure 4. Specifically, *any program state which is accessed by multiple threads must either be mapped as an AnvilJ Shared Instance, or be located on the same node as all its accessor threads*.

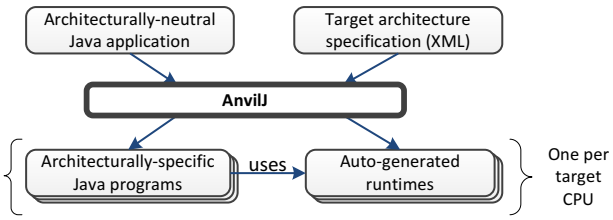

Fig. 5. The AnvilJ flow

Program state is defined as:

- The fields of an object instance
- The static fields of a class
- The state of an executing thread
- Object locks (from use of the `synchronized` keyword)

'Accessing' occurs when an instance invokes another instance's (static) method, (static) fields, or constructors. It can be seen that a sufficient way to guarantee the constraint is to ensure that all data shared between threads is marked as an AnvilJ Instance. Identifying and controlling the mutually-exclusive use of shared data for the elimination of race conditions is a difficult but well-understood problem in parallel programming. The use of analysable language subsets, model-driven development, coding guidelines and similar software engineering techniques mean that in practice the problem is tractable.

This is not a necessary solution, i.e. instances may be unnecessarily modelled, but because the AnvilJ runtime is very lightweight this does not introduce an unacceptable overhead. Calls to AnvilJ Instances from threads on the same node incur no extra overhead at all (the refactoring process simply does not invoke the runtime) so the only cost is that an extra entry in the shared method table of the AnvilJ runtime. In practice this is around 20 bytes, depending on the number of arguments.

In the running example of section III-D, `w1`, `w2`, and `Main` are declared as AnvilJ Threads and `wq` is declared as an AnvilJ Shared Instance.

IV. Implementation

The AnvilJ implementation flow is shown in figure 5. The input to the flow is a single, architecturally-neutral Java program, written under the restrictions of section IV-A, and an architecture specification file (section IV-G) which describes the input architecture and maps elements of the input program over it. The result of the process is one architecturally-specific Java program for each node of the system which is refactored to use its own custom-built, minimal runtime library.

After refactoring and code generation, the output programs have to be built to execute on the actual hardware. Discussion of this is outside the scope of this paper but can involve a special JVM for a target processor or a system such as Perc Pico [3] to compile the output Java to native code. The developer will also have to add hardware drivers, as detailed in section IV-D.

Currently, AnvilJ operates on applications written in standard Java so that AnvilJ is applicable to general embedded

programming, and because it is well understood by the industrial partners in the MADES project (see section V which is important for evaluation of the work. However AnvilJ will also support programs written using the RTSJ. The method and implementation remain broadly the same as discussed in this paper. As the RTSJ's `RealtimeThread` and `NoHeapRealtimeThread` extend `java.lang.Thread` they are already handled by AnvilJ. (Remember that the real-time JVM performs the more complex scheduling and thread dispatching of the RTSJ - AnvilJ merely has to refactor the code to use the real-time JVM.) Support will have to be added for the RTSJ's high-resolution time classes, event handlers, and the physical memory features as these require hooks adding to the runtime. These extensions are left for a future paper, although memory is discussed in section IV-H.

The rest of this section details the implementation of AnvilJ. Section IV-A enumerates the restrictions that AnvilJ places on its input Java code. Section IV-C then describes the AnvilJ refactoring engine whilst section IV-B describes the runtime generated by AnvilJ. The generated libraries are then detailed, and the Architecture Specification files described in section IV-G.

A. Code Restrictions

The following restrictions must be observed by AnvilJ's input code to ensure that communications occur at statically-defined points and predictability can be maintained. Code that does not communicate with other nodes can use all features of the language. As previously mentioned, a more general solution may remove the `static final` constraints and use an 'elaboration phase' that sets up the AnvilJ Instances. This was not done because in general a more restrictive and predictable Java subset (such as Ravenscar Java [14]) will be used, in which case many of these are already effectively mandated.

AnvilJ Instances must be `static final` fields
Makes the instances compile-time static and ensures they have a static initialiser, thereby allowing the refactoring engine to determine the expression used to create it.

AnvilJ Instances must be accessed by direct static reference
Use dot notation (`package.class`) if the instance is in another class. It is forbidden to 'leak' a reference to an AnvilJ Instance (e.g. returning it from a method, passing it to a method, or assigning it to a field). These actions are checked and prevented.

Arguments and return values of shared methods must be serialisable
The arguments and return values of shared methods that are exported by an AnvilJ Instance must be serialisable by the Java Object Serialization API [18].

AnvilJ also requires that fields of AnvilJ Instances are accessed through method calls, but it will automatically-generate 'getter' and 'setter' methods if required.

The first two restrictions could have been removed through the use of 'wrapper classes'. AnvilJ could inject new classes into the system which replace the classes of shared instances but call the runtime when their methods are accessed. This would remove the requirement for the refactoring engine to determine every shared instance reference, but as it would result in less predictable runtime behaviour the decision was taken to impose this restriction.

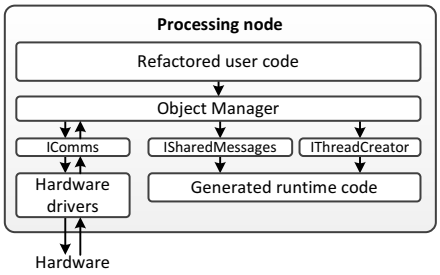

Fig. 6. Structure of the AnvilJ runtime

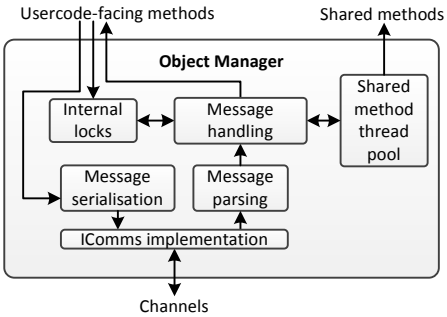

Fig. 7. Structure of the AnvilJ Object Manager

B. AnvilJ Runtime

As with all CTV-based systems, AnvilJ's runtime is composed of a set of interacting Object Managers (OMs). An OM is a microkernel (written in Java) that implements a minimal set of features that would normally be provided by a distributed OS. Full details of the OM model can be found in existing CTV literature [11]. After refactoring, each node in the target architecture contains exactly one OM. The OMs of the system exchange messages to implement the following features:

- Minimal distributed threading support (start, stop, join etc. AnvilJ Threads located on other JVMs)
- Remote method calls to allow threads of a node to invoke methods from AnvilJ Instances
- Remote monitor locks and thread synchronisation
- Routing of messages (to enable universal communications on non-uniform network topologies)

The runtime can be split into two sections; code which is common to all nodes, and code which is automatically-generated and specific to the exact node and the code that is mapped on to it. The runtime structure of the system is shown in figure 6. Three Java interfaces define the boundary between common code and generated code; `IComms` (section IV-D), `ISharedMessages` (section IV-E) and `IThreadCreator` (section IV-F).

The structure of the OM itself is shown in figure 7. The OM exposes a set of usercode-facing methods (which are invoked by the refactored input code) to send messages to other nodes, invoke remote shared methods, lock remote objects etc. Alongside this is a message processing thread which handles all messages that arrive from the hardware channels attached to this node. Most of these messages are handled directly by the handling thread (as they do not block). The two types of message handler that may block are messages to create AnvilJ threads and to invoke shared methods. When these arrive, they are passed to an internal thread pool for execution so that message handling cannot be delayed.

C. Refactoring

AnvilJ is implemented as a set of Eclipse plugins, which allows it to use proven Java parsers, analysis tools, and code generators. Full details of the refactoring process are outside of the scope of this paper but more details and examples of the refactoring are in the following sections.

D. The `IComms` Interface

The `IComms` interface defines the interface between the OM and the low-level hardware drivers. It provides methods to send and receive messages from other nodes. Routing is implemented using Dijkstra's algorithm with the generated runtime of each node containing static routing information. AnvilJ cannot automatically generate hardware drivers, so it is up to the developer to provide an implementation of `sendMessage()` which correctly uses the low-level channels, for example:

```
synchronized public void sendMessage
  (ObjectManager om, Message message) {
 switch(om.omIDtoChanID(message.targetOM)) {
  case 0: sendOverMailbox(message); break; //Driver
  case 1: sendOverEthernet(message); break; //Driver
   ...
```

The `omIDtoChanID` method of the OM maps target OMs to the channel used to communicate with them so that drivers can be independent of architecture, simplifying driver development. AnvilJ provides a default Berkeley sockets-based implementation of `IComms` which works on any OS that provides sockets and is useful for functional testing and debugging.

E. The `ISharedMessages` Interface

The `ISharedMessages` interface allows the OM to communicate with generated runtime code for executing shared methods. When an OM receives a message to execute a shared method from elsewhere in the system it must correctly unpack the arguments from the input byte stream, invoke the appropriate method, and return the result.

`ISharedMessages` contains data structures which describe the arguments and return types for all shared methods of all AnvilJ Instances located on this node. These structures are initialised in a static initialiser in the generated code for each OM, as the following output for the running example of section III-D shows:

```
static {
 ...
 //AnvilJ shared instance "wq"
 temp=new HashMap<Integer, ArgType[]>();
 //getWork
 temp.put(0, new ArgType[] { });
 //putWork
 temp.put(1, new ArgType[]
   { ArgType.INTEGER, ArgType.SERIALIZED });
 argumentMap.put(0, temp);
 ...
```

Generation of the return types data structure is similar. Support is built in to the refactoring framework to pack and unpack Java's native types to byte streams for transmission over channels. For other types, Java's serialisation API is used. Serialisation can result in a large amount of traffic so explicit support is added for a few commonly used classes (such as `java.lang.String`) and the user can also add their own serialisation routines.

`ISharedMessages.messageReceived()` is invoked by the runtime when a shared method is invoked on this node from elsewhere. An initial pass of the input code is made to ensure that all shared methods are `public` and so can be called. (In the running example `WorkQueue#getWork` and `WorkQueue#putWork` are altered.) This does not break visibility rules as the input program must be correct before refactoring takes place. A fragment of `messageReceived` from the running example is as follows:

```
public Object messageReceived(int SOID,
  int msgID, Object[] args) throws Throwable {
 switch (SOID) {
 case 0:
  switch (messageID) {
   case 0: return main.wq.getState();
   ...
```

Exceptions in shared methods are handled by the host OM sending a return message which indicates an unhandled exception and contains a serialised exception instance. This is deserialised by the calling OM and thrown so that it is caught in the user code as expected.

In the input code, calls of the shared methods are replaced as in the following fragment from the running example:

```
//Input code
Work w = wq.getWork();
//After refactoring
rv = (Work) anvilj.SharedMessageStubs.
  main_wq_getWork(0, 1, 0, new Object[] {}));
```

The `SharedMessageStubs` package contains a generated stub for each method which sends 'invoke remote method' messages to the host node of the target shared object with the appropriate arguments passed in an Object array.

F. The `IThreadCreator` Interface

The `IThreadCreator` interface allows the OM to invoke the generated code which creates and starts the AnvilJ Threads on its node, when instructed to by remote nodes. The refactoring finds the initialisation of the AnvilJ Thread (required because it is `static final`) and then adds a new method in the host class which performs the same operations and returns the created instance.

```
<architecture name="Runningexamplearch"
    mainclass="main.Main" maincpuid="0">
 <cpu name="CPU0" id="0">
   <thread binding=
     "Lmain/Main;.w1)Lmain/Main/Worker;"/>
   <sharedobject binding="Lmain/Main;.wq;"/>
 </cpu>
 <cpu name="CPU1" id="1">
   <thread binding=
     "Lmain/Main;.w2)Lmain/Main/Worker;"/>
 </cpu>
 <channel name="chan0">
   <endpoint cpu="CPU0"/> <endpoint cpu="CPU1"/>
 </channel>
</architecture>
```

Fig. 8. Architecture Specification for the running example

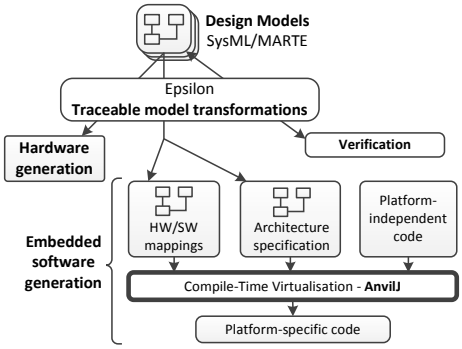

Fig. 9. Overview of the artifacts in the MADES approach

```
//Input code (from running example)
final static Worker w1 = new Worker();
//Refactored method
public static Worker anviljCreate_w1() {
 return new Worker();}
```

These generated methods are then called from a generated stub as part of the `IThreadCreator` implementation:

```
public Thread createThread(int threadID) {
 switch (threadID) {
 case 0:
  return Main.anviljCreate_w1();
  ...
```

`IThreadCreator#createThread` can then be used by the OM to create threads in response to a 'create thread' message from remote nodes. On all other nodes the refactoring engine removes the `static final` instance `w1` (in the case of the example) and replaces any old calls to `w1.start()` with:

```
anvilj.om.startThread(0, 1); //(node 0, threadID 1)
```

`startThread` sends the appropriate 'create thread' message to the target node using the `IComms` interface. Other references to `w1` are similarly replaced.

G. Architecture Specification

AnvilJ uses XML-based Architecture Specifications. Figure 8 shows the architecture specification for the running example of section III-D.

The main element of the specification is `architecture` which has attributes to provide a name, identify the main Java class in dot notation (`Class.Package`) and map the main thread to a node. `cpu` elements model processing nodes and have `name` and `id` attributes, both of which must be globally-unique. `thread` and `sharedobject` elements must have a `binding` attribute which associate the thread or shared object with a `static final` instance in the input application. Bindings uniquely identify classes, fields, and methods in input source code. They are defined in the Eclipse JDT project [22] and a full description is outside the scope of this paper. AnvilJ's tooling provides the designer with easy ways to obtain the binding keys of source code elements.

`channel` and `memory` elements must contain at least one `endpoint` element that defines the connections between the channel or memory and the CPU nodes of the system. `endpoint` nodes contain a mandatory `cpu` attribute which is the name of the `cpu` element to which they are attached. AnvilJ checks that these attributes are all correctly set.

Extra information may be added to any node using `attr` elements. This information is not required for the base AnvilJ implementation, but is made available to drivers (see section IV-D).

H. Shared memory

As detailed in section III-B, the AnvilJ system model allows the definition of shared memory areas into which AnvilJ Shared Instances may be mapped. This is implemented differently according to the input language. Java using standard JVMs does not natively support multiple memory spaces. Therefore when Java is used such a mapping must be reduced to simply map the instance to one of the nodes that are connected to the shared memory. The refactoring engine warns the user that such mappings are not fully supported with this input language. Functional correctness is maintained, however.

However, the RTSJ allows the specification of more complex memory architectures with its physical memory framework which allows the programmer to place objects in specific memory locations and implement shared memory. Recent work [15] has shown how such features can be used to control memory access in a non-uniform memory architecture. As discussed in section III-B, AnvilJ does not support refactoring RTSJ code at the time of writing, but this is ongoing work.

V. Evaluation

AnvilJ is developed as part of MADES [1], an EU Seventh Framework project which is developing a fully model-driven approach for the design, validation, simulation, and code generation and deployment of systems. In the MADES framework, high-level system models (in a combination of SysML [25] and MARTE [17]) provide information about the target architecture and the structure of the input software. UML deployment diagrams are used to provide virtualisation mappings. These models and diagrams are transformed in traceable and certifiable steps from initial specification to the final deployed system using the Epsilon model transformation tools [13].

MADES incorporates three parallel toolchains for performing hardware development, temporal verification/validation, and software development (of which AnvilJ is the main component, shown in figure 9). All these aspects, including AnvilJ, will be fully validated on real-life case studies in the surveillance and avionic domains by MADES' industrial partners. The evaluation in this paper focuses on the predictability of the AnvilJ output programs, and the size of the overhead introduced by AnvilJ's runtime.

A. Preservation of predictability

An important requirement of AnvilJ is that the refactoring engine preserves the latent predictability of the input code, meaning that AnvilJ does not introduce unpredictable behaviour. To evaluate this it is necessary to examine the translations that AnvilJ makes when transforming a single processor shared memory model to a message passing model.

Runtime

The AnvilJ runtime (section IV-B) observes the predictability assumptions of the input. The majority of user-facing methods are straight-line code with no blocking so their execution time can be bounded. Only three features are not, and must be considered separately.

Firstly, user code invoking the OM to obtain a remote monitor lock can cause blocking, but this is bounded according to the input assumptions of the system (section III-C). Network overhead is added to remote locks, but again this is assumed to be bounded by the input model. The refactoring engine only inserts such invocations when the user uses a `synchronized` statement, or calls a remote `synchronized` method. AnvilJ's static programming model means that it is always possible to assess when input code assess a remote AnvilJ Instance, therefore the effect of this is bounded.

Second is the OM's message handling thread which must be added to the analysis of the input system and modelled. By design, all AnvilJ message handlers cannot block (apart from the two discussed below) so their effect on the node is bounded by the communications to that node, which is also bounded from the input assumptions.

Finally, the thread pool for executing remote shared methods or remote thread initialisation. From the input assumptions it is possible to bound the computation due to remote invocations of any AnvilJ Instance's shared method. This computation, which was previously located on the accessor node must be modelled as computation on the host node. Also, it is necessary to demonstrate that the thread pool is of sufficient size such that a thread is always available when a shared message is invoked. A trivial worst case for this is obtained by summing the total number of threads that may call each shared method for the node. The assumptions state that the input code may be bounded, so this number of threads can be also. This naive test may be pessimistic; and further work will refine it.

Generated runtime

All methods of the generation runtime are straight-line code made of switch statements and stub methods. The predictability of the system however is reliant on a real-time network or communications infrastructure being available on the platform which exhibits bounded timing behaviour, which is outside the scope of AnvilJ's analysis.

Output code

The output code is identical to the input code apart from a few well-defined points. First, all invocations of AnvilJ Instance methods which are located on a remote node are refactored to send a message. Due to AnvilJ's static model these locations can be determined offline. Complexity occurs when a node is used as part of a multi-hop route. Then, time must be provisioned on both the sender, receiver *and* the intermediary, however these routes are calculated offline. The other part of the input code which is altered to inject communications is when a `synchronized` statement is used to obtain the lock of a remote AnvilJ Instance.

It is worth reiterating that AnvilJ does not guarantee a schedulable output system, as it can be used to target architectures which are so complex that their analysis is infeasible. However, this section argues that if the input code follows the assumptions in section III-C then the effect of the translation is bounded and the code is made no less predictable.

Equally, the assumptions made in this paper assume that AnvilJ generates the only application on the target hardware. It is further work to analyse the effect of executing an AnvilJ applications alongside other application on the same hardware under real-time constraints.

B. Overheads

The AnvilJ runtime only imposes small overheads, especially when compared with much larger (although more powerful) general-purpose frameworks. The main overhead in an AnvilJ system is that of the OM; the full version of which compiles to approximately 34kB of class files including debugging and error information. It is also possible to create smaller OMs which only support a subset of features for when the software mapped to a node does not require them. The advantage of AnvilJ's system model and offline analysis is that this can be done automatically each time, based on the exact input application and hardware mappings. Figure 10 shows a breakdown of some of the feature sets of the OM and their respective code footprint.

Feature set	**Approx. size**
Thread creation and joining	5.7 kB
Remote Object Locks	4.5 kB
Shared methods	8.4 kB
Sockets-based IComms (debug)	4.29 kB
Full OM	34 kB

Fig. 10. Class file sizes for OM features

In addition to the small code size of the OM, its runtime memory footprint is also modest. Measurements vary between JVM, OS, and target system, but the full OM in a desktop

Feature	Approx. size
Full OM	648 bytes
Hosted Thread	32 bytes
Hosted Shared Instance	32 bytes
Each hosted shared method	20 bytes
Sockets-based IComms (debug)	360 bytes

Fig. 11. Dynamic memory footprints for OM features

Linux-based system uses approximately 648 bytes of storage when idle, which increases as clients begin to use its features. Dynamic sizes are shown in figure 11. In a real-time implementation, all dynamic memory (queues, buffers, etc.) can be capped at a fixed size that is determined by the designer and offline analysis.

Clearly when an AnvilJ Instance is shared on a remote node, accessing it will involve the transfer of messages and this could be considered an overhead. However, in such a system this is unavoidable. Exploring system mappings to minimise such transfer is outside of the scope of this paper.

It is not appropriate to attempt to directly measure performance overheads because no fair comparison can be made. CTV-based systems achieve very low overheads by sacrificing runtime flexibility. Systems like CORBA or Teamster [8] do the opposite, achieving flexibility but at the expense of a larger impact on the system. However for reference, most Real-Time CORBA ORBS are larger than 2000kB in static code size but are still used for embedded systems, and embedded Linux kernels (such as uClinux) whilst varying greatly are often between 900kB and 1400kB. Perc Pico [3] (a real-time Java runtime based on translation to C) requires 256-512kB of memory. It can therefore be said that AnvilJ's overheads are relatively small.

VI. Conclusion

This paper has described AnvilJ, a novel approach for the development of software for non-standard, embedded architectures in Java. Most general-purpose systems that are designed to aid the development of software for complex architectures operate primarily at runtime. They are very flexible, but must support the full variability of the input language and target architecture. AnvilJ operates primarily at compile-time and uses a restricted system model based on a technique called Compile-Time Virtualisation. This restricted model allows AnvilJ to operate with vastly reduced runtime support that is predictable and bounded.

The paper has described how the implementation extends to use the Real-Time Specification for Java, and focuses on the development of a predictable system such that neither its compile-time nor runtime transformations introduce non-determinism. Unlike most existing runtime frameworks, the runtime and supporting libraries are generated during compilation for each specific combination of input program and target architecture. This ensures minimal overheads and that the final system exhibits static, analysable behaviour.

When programs are written according to a defined set of input assumptions concerning predictability and worst-case behaviour, this paper argued that after the AnvilJ translation the final system will be equally analysable. Predictability is preserved, making AnvilJ suitable for the development of real-time embedded systems.

References

[1] The MADES Project. http://www.mades-project.org/, 2011.

[2] J. Andersson, S. Weber, E. Cecchet, C. Jensen, and V. Cahill. Kaffemik – A distributed JVM on a single address space architecture, 2001.

[3] Atego. Perc Pico. http://www.atego.com/products/aonix-perc-pico/, 2011.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02*, pages 73–78, 2002.

[5] Baumann et al. The Multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of SOSP '09*, pages 29–44, New York, NY, USA, 2009. ACM.

[6] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. In *Ada-Europe '98*, pages 263–275. Springer-Verlag, 1998.

[7] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[8] J.-B. Chang, C.-K. Shieh, and T.-Y. Liang. A transparent distributed shared memory for clustered symmetric multiprocessors. *The Journal of Supercomputing*, 37(2):145–160, 2006.

[9] K. Fatahalian et al. Sequoia: programming the memory hierarchy. In *SC '06*, page 83, 2006.

[10] J. Gosling and G. Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[11] I. Gray and N. Audsley. Exposing non-standard architectures to embedded software using Compile-Time Virtualisation. *CASES '09*, 2009.

[12] I. Gray and N. Audsley. Targeting complex embedded architectures by combining the multicore communications API (MCAPI) with Compile-Time Virtualisation. In *LCTES'11*, 2011.

[13] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Eclipse development tools for epsilon. In *In Eclipse Summit Europe, Eclipse Modeling Symposium*, 2006.

[14] J. Kwon, A. Wellings, and S. King. Ravenscar-Java: A high integrity profile for Real-Time Java. In *In Joint ACM Java Grande/ISCOPE Conference*, pages 131–140. ACM Press, 2002.

[15] A. H. Malik, A. Wellings, and Y. Chang. A locality model for the real-time specification for Java. In *JTRES '10*, pages 36–45, New York, NY, USA, 2010. ACM.

[16] P. Marwedel. *Embedded System Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[17] Object Management Group. UML profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. http://www.omgmarte.org/, November 2009.

[18] Oracle. Java object serialization specification v 1.5.0. http://download.oracle.com/javase/1,5.0/docs/guide/serialization/, 2004.

[19] F. Pizlo, L. Ziarek, and J. Vitek. Real Time Java on resource-constrained platforms with Fiji VM. In *Proceedings of JTRES*, JTRES '09, pages 110–119, New York, NY, USA, 2009. ACM.

[20] A. L. Pope. *The CORBA reference guide: understanding the Common Object Request Broker Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[21] Terracotta Inc. *The Definitive Guide to Terracotta - Cluster the JVM for Spring, Hibernate and POJO Scalability*. Apress, 2008.

[22] The Eclipse Foundation. Eclipse Java development tools. http://www.eclipse.org/jdt/, 2011.

[23] The Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Critical Systems*. MISRA Ltd., 2004.

[24] W. Thies et al. StreamIt: A compiler for streaming applications, December 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.

[25] T. Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.