Australian Government

Department of Defence
Science and Technology

A History of Jaca's Progress to a Field Programmable Gate Array

Jeffrey W. Tweedale

Weapons and Combat Systems Division
Defence Science and Technology Group

DST-Group-TN-1723

ABSTRACT

This report documents a literature review of the methods associated with embedding a JVM into a Field Programmable Grid or Gate Array (FPGA). The review demonstrated that there was originally a commercial interest in providing silicon solutions for embedded JVMs, however with the exception of the JavaCard (which is embedded in all credit cards), few actually survived as mainstream products. To fabricate this form of design the developer requires a highly multi- disciplined team, that includes skills in programming, OS design, FPGA development and a signicant experience with very-low level hardware design. This report was written primarily to explore the existing domain by documenting the history, developments and possible future for this form of technology. A description of the basic concept together with more advanced applications have been described as example uses of this technology. Ultimately Defence Science and Technology Group (DST Group) could exploit the reuse and exibility of this approach to interface with legacy systems or exploit their interoperability for Defence.

Published by

Weapons and Combat Systems Division
Defence Science and Technology Group
PO Box 1500
Edinburgh, South Australia 5111, Australia

Telephone: *1300 333 362*
Facsimile: *(08) 7389 6567*

AR-017-071
January, 2018

APPROVED FOR PUBLIC RELEASE

Executive Summary

As the Java programming language continues to mature, the rhetoric about its effectiveness in the enterprise environment diminishes. Ultimately, this sentiment softens with each new release of its specification. Historically Java was supported through academia and has been adopted by enthusiasts, who willingly engage in open source projects. It has become a language of choice for many large corporations because it is secure, robust, reliable and type safe. Being platform independent, it is especially useful for transactional and network based applications. Given the future of parallel processing and the growing use of intelligent agent teams within complex programs, Java is a natural language of choice. Although languages like Erlang, Closure and Scala are being promoted as the future, Java still has a significant following and is supported by industry, academia and a growing crowd-sourced communities. Even if Scala is chosen in the future, its bytecode can execute on a Java Virtual Machine (JVM).

The work presented in this report provides details of how researchers have adapted the use of a Field Programmable Grid or Gate Array (FPGA) design to embed a Java Virtual Machine as a logical solution in silicon hardware. A literature review shows there was some initial commercial interest in providing silicon solutions. This paper was written primarily to explore the existing domain and document the history, developments and possible future for this form of technology. Existing concepts are discussed and the document delineates how this capability could be used to enable the proliferation of common interfaces that promote seamless interoperability to the wider defence community.

The most recent open source effort in this domain appeared within academia in Europe as the Java Optimized Processor (JOP). To fabricate this design, the developer requires a highly skilled, multi-disciplined team, to overcome design issues at many levels of the computing architecture. This starts from the application layer and extend through to middleware, Operating Systems (OSs), kernel and even into the physical hardware. Members of the team need specific skills in FPGA and very detailed knowledge of hardware design.

This review also outlines the development of the data structures and algorithms used to support running software in silicon using FPGAs. This means executing bytecode on a physical JVM using a hardware solution. Existing methods allow developers to run native processors that directly interpret bytecode without the need for emulation or Just-in-Time (JIT) compilation. A customised array of physical JVM cores designed to interoperate within silicon is envisaged to reduce design complexity and massively enhance its deterministic performance. Using a services approach for connectivity will also enable the seamless approach to Network Centric Warfare (NCW) on a massive scale at low cost. This concept could be used to rapidly create interfaces to legacy systems, exploit interoperability options that are currently not feasible and even leverage the functionality provided through enterprise applications in the field. This deterministic solution has potential uses in mission critical applications and further investigation is recommended.

THIS PAGE IS INTENTIONALLY BLANK

Author

Jeffrey W. Tweedale
WCSD

Dr Jeffrey Tweedale is a Scientific Engineering Officer who works at the Tactical Systems Integration Branch of the Weapons and Combat Systems Division within the Defence Science and Technology Group in Australia. He has been an adjunct member of the Knowledge-Based Intelligent Engineering Systems (KES) Centre at the University of South Australia since 2004. He earned a PhD in Computer Science Engineering from the University of South Australia, an MBA in Business from the Adelaide University, B. Comp (Hons) degree in Computer Science from Monash University, a B.IT in Information Systems from Charles Sturt University and a B. Ed in TAFE from Melbourne University. His recent research interests include: Multi-Agent Systems, Unmanned Aerial Vehicles, System Automation, Human-Computer Trust, Electronic Engineering and Computer-Based Learning. His current research involves maritime computer vision and autonomous airborne platforms using distributed sensor networks.

THIS PAGE IS INTENTIONALLY BLANK

Contents

Appendices

Figures

THIS PAGE IS INTENTIONALLY BLANK

Acronyms and Abbreviations

ACE Adaptive Communication Environment

ACL Access Control List

ACL2 Access Control List language - Version 2

ADF Australian Defence Force

AEW&C Airborne Early Warning and Control

ALU Arithmetic Logic Unit

AMD Advanced Micro Devices

AOT Ahead of Time

API Application Program Interface

ARM Advanced RISC Machine

ASIC Application-Specific Integrated Circuit

ASML Advanced Semiconductor Materials Lithography

ASIC Application-Specific Integrated Circuit

ATM Asynchronous Transfer Mode

BIOS Built-in Operating System

BluEJAMM BlueSpec Embedded Java Architecture with Memory Management

CDC Connected Device Configuration

CDI Context and Dependency Injection

CISC Complex Instruction Set Computer

CLB Customised Logic Blocks

CLDC Connected, Limited Device Configuration

CLB Configurable Logic block

CMP Chip Multi-Processing

CIOP Control Area Network Inter-ORB Protocol

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

DARPA Defense Advanced Research Project Agency

DBX Direct Bytecode eXecution

DCE Distributed Computing Environment

DCOM Distributed Component Object Model

DDS Data Distribution Service

DSP Digital Signal Processor

DST Group Defence Science and Technology Group

EEPROM Electrically Erasable Programmable Read-Only Memory

EFTPOS Electronic Funds Transfer Point-Off Sale

EJB Enterprise Java Beans

EUV Extreme Ultra-Violet

FIPA Foundation of Intelligent Physical Agents

FPGA Field Programmable Grid or Gate Array

FPU Floating-Point Unit

GIOP General Inter-ORB Protocol

GPU Graphics Processing Unit

GUI Graphical User Interface

HDD Hard Disk Drive

HDL Hardware Description Language

HISC High-Level Instruction Set Computer

HPC High Performance Computing

HTML HyperText Markup Language

ICT Information and Communications Technology

IIOP Internet Inter-ORB Protocol

ILP Instruction Level Parallelism

I/O Input/Output

IP Internet Protocol

ISA Instruction Set Architecture

ISE Integrated Synthesis Environment

J2EE Java 2 Enterprise Edition

J2ME Java 2 Micro Edition

J2SE Java 2 Standard Edition

JAX-RPC Java API for XML-based RPC

JCP Java Community Process

JCRE JavaCard Run-time Environment

JDK Java Developers Kit

JINI Java Intelligent Network Interface

JIT Just-in-Time

JMS Java Message Service

JMX Java Management eXtensions

JNI Java Native Interface

JOP Java Optimized Processor

JSM Java Silicon Machine

JSON JavaScript Object Notation

JSR Java Specification Requests

JTAG Joint Test Action Group

JTRES Java Technologies for Real-time and Embedded Systems

JVM Java Virtual Machine

KQML Knowledge Query Manipulation Language

KVM kilo-Virtual Machine

LAN Local Area Network

LBIST Logical Built in Self-test

LC Logic Cell

LED Light Emitting Diode

LFSR Linear Feedback Shift Register

LFSR Linear Feed Shift Register

loc Lines of Code

NCW Network Centric Warfare

MIDP Mobile Information Device Profile

MISC Minimum Instruction Set Computer

MISR Multiple Input Shift Register

MVC Model-View-Controller

NeWS Network extensible Window System

OMG Object Management Group

OOP Object Oriented Programming

OOPL Object Oriented Programming Language

ORB Object Request Brocker

OS Operating System

OSI Open Systems Interconnection

PC Personal Computer

PDA Personal Digital Assistant

PID Proportional Integral Derivative

PIN Personal Identification Number

POSIX Portable Operating System Interface

RPC Remote Procedure Call

RAM Random Access Memory

REST RESTful web framework for Java

RISC Reduced Instruction Set Computer

ROM Read Only Memory

RPC Remote Procedure Call

RTOS Real-Time Operating System

RTSJ Real-Time Specification for Java

RSoC Reconfigurable System on Chip

RTL Register Transfer Level

SMT Simulataneous Multi-threaded

SOAP Simple Object Access Protocol

SOC System on Chip

SRAM Static Random Access Memory

SRSG Shift Register Sequence Generator

STI Sony, Toshiba, and IBM

TCP/IP Transmission Control Protocol / Internet Protocol

TDMA Time Division Multiple Access

TAO The Ace ORB

TOS Top of Stack

TSMC Taiwan Semiconductor Manufacturing Company

UAV Unmanned Air Vehicle

UCSD University of California, San Diego

VHDL Very High Speed Integrated Circuit Hardware Description Language

VLIW Very Long Instruction Word

VM Virtual Machine

WAN Wide Area Network

WCET Worst Case Execution Time

WSRP Web Services for Remote Portlets

XML Extensible Markup Language

XOM XML Object Model

THIS PAGE IS INTENTIONALLY BLANK

1 Background

In computer-based systems, a hardware technology refresh is often used to overcome complex problems. Designers can often achieve better, smarter and cheaper products by using Field Programmable Grid or Gate Arrays (FPGAs). This technology breaks the cost barrier typically associated with fabricating Application-Specific Integrated Circuit (ASIC) devices. The use of FPGAs also speeds up the development of silicon solutions. Software-defined silicon solutions are becoming mainstream. The public supply of System on Chip (SOC) solutions are also becoming reality. This is enabled by the host of tool chains in the FPGA domain. Software is becoming firmware and the boundary between programs and hardware is increasingly becoming blurred. A silicon solution can now be used to hard-wire effective solutions, (such as code folding, variable stack access, using pipelines, caches, reduced instruction sets and even hardware-based garbage collection). Experience will enable programmers to write software which uses less resources and hence operate more efficiently, for example by using compound statements or embedding variables within a function state, where code can include the data within the stream and execute in a single cycle. This concept is explored further in Section B.2.

As Java evolved, interest in native designs faded [1] and industry started focussing on more convenient software solutions, such as the Just-in-Time (JIT) compiler. Hence the public supply of SOC solutions, such as Java in Silicon, are now becoming reality. As a result, industry is able to implement technology refresh faster and with more certainty. Java in Silicon is one of the beneficiaries of this trend. Today technology allows increased performance to be achieved by the native execution of bytecode on dedicated processors using FPGAs. The same renaissance is also facilitating main stream researchers access to tools that enable either virtual or physical realisation of concepts traditionally impaired through lack of skills, money or time.

One of the reasons to focus on Java is its popularity across academia. Unless they specialise, most programmers now graduate with Java being their primarily programming language. It is now the mainstream programming language for teaching, hobbyists, enthusiasts and for many professionals supporting open source projects. Java has already displaced many existing languages and initially posed a threat to C++ [2]. While it has had a major impact within the enterprise environment, it continues to evolve as a ubiquitous service-based language for mobile, network-based and Internet applications. Major reasons for this include the ability to encapsulate entities that are type safe using polymorphism, dynamic binding and simple inheritance. Over time it has become increasingly robust because of its security features enabled through dynamic garbage collection. Exception handling, bounded arrays and constrained referencing also play important roles in security. The cumulative package enables portability across heterogeneous desktop platforms. With the proliferation of mobile devices, Sun Microsystems streamlined Java 2 Standard Edition (J2SE) with a targeted version of this technology using Java 2 Micro Edition (J2ME). Similarly an enterprise environment was also created to cater for corporate activities.

In the early literature on FPGA techniques used to embed a Java Virtual Machine (JVM) into silicon, there was limited evidence of commercial interest in hardware solutions, and only a scattering of academic effort [3, 4, 5]. Over the last decade research in this area has

picked up and it is concentrated in Europe. For example there were over 80 papers published during 2011. Recent work on Object Oriented Programming Languages (OOPLs) and subsequent effort on intelligent agent research reinforced the need for an alternative approach to using the traditional *harvard* architectural model when implementing a *Turing Machine* [6]. At present we make the software fit the machine, however Java and the JVM were designed specifically to cater for Object-Oriented software [7]. This concept was originally aimed at completely abstracting away the physical machine. A publication by this author has shown how viable alternatives to software solutions can be created in hardware that can easily achieve more flexible and less complex designs using Java [8]. This document further elaborates on how hardware can obtain significant benefits by combining Java and the FPGA technology.

2 Aim

The aim of this research is to review the literature and illustrate how embedding a JVM into silicon using an FPGA can potentially provide a longer-term benefit to Defence.

3 State of the Art in Java

All programming languages have a predefined syntax that is used to generate code that represents a higher-level depiction of the final executable software (regardless of being compiled or interpreted). Languages have evolved with respect to the environment and problem space generating the solution. Machine code, Cobol and Pascal are classic examples. Appendix A provide some background on how Java evolved as a state of the art programming language. The main topics discussed include: the concepts associated with data structures, bytecode, target applications, pedigree, the current release, the diversity as a host language and the concept of using Java syntax as an interpreter to reuse code or algorithms written in other languages. This information has been generated from the literature review to provide supporting facts and the status of the current specification of the JVM.

The concepts associated with interpreting programming languages using virtual machines are not new. The 'Basic' programming language is a classic example of software that employed the use of intermediate code that is mediated prior to execution[1]. The concept of platform independence used in Java is also borrowed. It was demonstrated in the early 1970's using SmallTalk and later with Pascal (specifically University of California, San Diego (UCSD) Pascal *p-code* [10]). Before examining the technology trends it is important to acknowledge that Java benefited from the evolution of other languages.

[1] An image of an application in byte-code is far smaller than traditional executable applications. This reduces bandwidth and storage requirements and is obviously more efficient in a networked environment [9].

3.1 Technology Trends

New technologies frequently appear on the horizon, however it is difficult to determine whether they pose a disruptive threat in our domain. We know that today's Central Processing Units (CPUs) do not provide an efficient means to solve all problems. Not long ago the Information and Communications Technology (ICT) community viewed the *microprocessor as the machine*. With the introduction of Local Area Networks (LANs) and Wide Area Networks (WANs), the *network became the computer*[2]. With the onset of mobile computing, we increasingly consider the *cloud as the computer*.

Our need for more complex computation has stimulated the growth of super computers using massively parallel techniques. High Performance Computing (HPC) functionality is generally created, accessed and operated within a facility dedicated to a predefined task. This changed when the Sony, Toshiba, and IBM (STI) alliance introduced the *Cell* microprocessor. According to the Lawrence Berkeley National Laboratory, this delivered the power of the original *Cray* super computer to a games console in the home [12]. This growth of processor throughput and functionality evolved to deliver more complex capability to the wider community. The introduction of multi-core microprocessors with multi-chip platforms also delivered more applications into the field. This trend is increasing as consumers demand more capability on their phones. The pitfall for Defence using this approach is in providing the interoperability, bandwidth, security and reliability (which indicates the possible need for a private cloud).

Specialised chips and architectures are increasingly being used to solve specific issues. In the real-world programmers customize software to achieve their goals, but in an ideal world designers would also like to be able to customise the functionality and operation of physical hardware (or at least the firmware). As technology migrates further along the path of parallel processing, we should be able to imagine the resources as simple components of the overall solution. Figure 1 displays the concept of what a customised computer could look like in 20 years. This would comprise an ensemble of designs on a single FPGA using a selection of processor types. Example design libraries currently include:

- Java Virtual Machine (JVM),
- Central Processing Unit (CPU),
- Graphics Processing Unit (GPU),
- Digital Signal Processor (DSP) and
- Configurable Logic block (CLB)

Clusters or arrays of processing units can be functionally assembled or individually customized to provide more efficient and solution focused configurations. This concept initially appears ambitious until the facts are reviewed. The first factor to consider is Moore's law [13]. Next are the technology drivers delivered through competition by Intel's *tick-tock*

[2]This phrase become the *motto* of Sun Microsystems for over forty years through its CEO Scott McNealy [11].

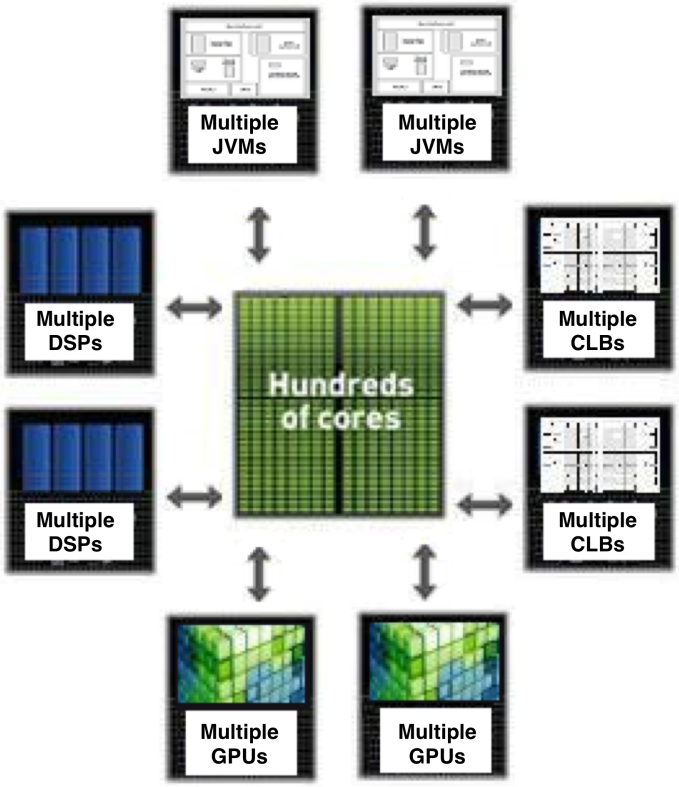

Figure 1: What will a Machine look like in 20 years

program and the evolution of lithography processes[3]. Both bear witness to the ongoing evolution in miniaturization with enhanced performance. Each time companies like Intel, Advanced Micro Devices (AMD), Samsung Taiwan Semiconductor Manufacturing Company (TSMC) or other manufacturers migrate to the latest generation of fabrication technology, industry delivers improved benchmarks, reduced power consumption and produced significantly higher chip densities. With the release of the 10 core Westmere microprocessor, Intel confirmed the viability of its 22 nm fabrication technique[4]. Recently AMD released their Zen CPU and Intel released their 6^{th} generation Skylake microprocessor to market, both using 14 nm technology. Intel are currently transitioning memory products to 11 nm and plan to transition to 8 nm in the immediate future[5]. Given this level of miniaturization, using 3D components and copper interconnects, single chip densities in the order of 100 billion transistors are realistically possible by 2020 (using the same amount of silicon we do today would produce approximately 250 *Skylake* cores on the same die.).

[3]Global Foundries reported it is installing Advanced Semiconductor Materials Lithography (ASML) hybrid 193-immersion Extreme Ultra-Violet (EUV) processes targeted at 5 nm by 2017 to allow other manufactures to release similar technology to market by 2018 [14, 15, 16].

[4]Intel actually pledged an 80 core single chip prototype within five years at the 2006 Intel Developers Forum [17] and demonstrated a 275 mm^2 network-on-chip architecture contains 80 4 GHz tiles of floating-point cores and packet-switched routers (arranged as a 10*8 2D array) at the International Solid-State Circuits Conference in 2007 [18]

[5]At 14 nm, 1.75 billion transistors are used for the Skylake processor, however 10 billion are theoretically feasible

In the not too distant future, microprocessors will be treated as components modelled around clusters (using quad, octal or even hexadecimal designs), on a single carrier, complete with interconnected networks of arbitrated buses[6]. This concept will facilitate dynamically regenerative homogeneous agents that are synthesized (virtually or in Silicon) automatically when instantiated or invoked[7]. In an ideal world this would enable the programmer to evolve concepts without the constraints currently experienced within the platform, in the network or imposed when simulating the environment.

3.2 Existing Contributions

The concept of embedding a Virtual Machine (VM) as a set of physical logic circuits is not new. Both industry and academia have pursued Silicon solutions to physically execute Pascal and Forth. This document focuses on Java and the ability to embed a physical JVM into silicon using FPGAs. Over the past decade a number of researchers have experimented with solutions using hardware acceleration, translation and even dedicated processors to run byte code as native instructions. Several of these are listed below:

Acceleration: A coprocessor is attached to the host and Bytecode is redirected and processed independent to the core software. Recently designs integrate the coprocessor with the core processor. These processors provide support without compromising host compatibility, but consume significantly more power. Examples include:

- AU-J2000 from Aurora VLSI[8],
- MOCA-J from NanoAmp Solutions[9], and
- Moon 2 from Vulcan Machines (Can be native or acts as a coprocessor)[10].

Translation: A small hardware unit is positioned just prior to the instruction cache to translate Bytecode into native machine code. These processors can constrain the use of some key features offered by Java, such as security, garbage collection and object scope at hardware level. Examples include:

- ARM Jazelle from ARM[11],
- JA108 from Nazomi Communications[12], and
- JVXTreme from Synopsys (previously Insilicon)[13].

Dedication: Bytecode is executed directly, similar to machine code on traditional processors, without the need for interpretation or translation. At present these solutions

[6] This issue involves increased complexity as explained by Metcalfes Law [19].

[7] Xilinx are already producing their series 7 products at 28 nm and increasingly integrating support peripherals to compliment the rich set of CLB currently provided. Future designs could feasibly see enhancement to the type and density of embedded components, making design more effortless and efficient.

[8] See http://www.auroravlsi.com/.

[9] See http://www.nanoamp.com/.

[10] See http://www.vulcanasic.com/.

[11] See http://www.arm.com/products/processors/technologies/jazelle.php.

[12] See www.nazomi.com/ and java.epicentertech.com/Archive_Embedded/Nazomi/ja108_pb.pdf.

[13] See www.synopsys.com/dw/doc.php/doc/smartmodel/manuals/simcfg.pdf.

are limited to simple implementations that are confined to a stack design. They are not able to execute applications with an alternate instruction set. Examples include:

- picoJava from Sun Microsystems[14],
- aJ-100 and aJ-200 from aJile Systems (This contains the JEMCore)[15], and
- lavaCore from Xilinx[16].

3.3 The Traditional Java Virtual Machine

A JVM is software that runs as a process within the Operating System (OS) to interpret bytecode files when required. The JVM interacts with the Built-in Operating System (BIOS) when resources are required for Input/Output (I/O) operations. The basic concepts associated with the traditional JVM are discussed in Appendix B.1. Regardless of the pedigree or platform, every native processor must decode and implement each instruction as a logical operation. Machine code instructions provide the logic required to configure the processor to execute the intended operation of each command. A queue of these instructions performs the function generated via the compiler. The logic associated with implementing the functions of a JVM using an FPGA can be extracted directly from the specification for the virtual machine. Appendix B.1 also describes how the Java instruction set supports the bytecode specification and memory model (as discussed in Appendix B.2 and B.3).

3.4 The Hardware Implementation of a Java Machine

A physical Java Machine (also labelled a Java Silicon Machine (JSM)) provides the logical circuitry required to directly execute bytecode instructions (without the need for cross-compilation or interpretation) and all traditional support logic associated with a traditional computer. This machine will contain the logic required to execute as a physical JVM (B.4). The logic circuits required can be easily assembled in a silicon solution[17]. The solution would be capable of directly executing a stream of bytecode as native machine readable instructions. As discussed, this concept is similar to using a math co-processor or GPU, although the physical machine discussed (which includes the JVM, memory management and I/O) is used as a direct alternative to the CPU, such as those provided by Intel and AMD.

As suggested, the evolution of the JIT compiler constrained the production of dedicated chips and support for silicon solutions faded. Given that Java has established its credentials as a mainstream programming language and has a deterministic operating system to support mission critical systems (See Real-Time Specification for Java (RTSJ) in Appendix B.5), it is increasingly being used by industry. Because of this support and the

[14] See http://www.oracle.com/.

[15] See http://www.ajile.com/.

[16] See http://www.xilinx.com/.

[17] Such as an FPGA using Hardware Description Language (HDL) connections to assemble Very High Speed Integrated Circuit Hardware Description Language (VHDL) component libraries (half adders, stacks, cache, memory, comparators and so forth).

ease of developing Java software, the focus on dedicated processing is re-gaining strength, especially in the embedded market.

Sun Microsystems was ambitious when it embarked on its three physical processor designs. Their vision was to alleviate any implementation or software incompatibility and provide a universal programming capability for all platforms. A number of industries expressed interest in this concept and continued investigating physical stack based designs. Two examples include *Patriot* and *ShBoom* [20]. Early research using Java-chip simulations pointed to significant speed advantages (a multiple of 11.3) over general-purpose CPUs using emulated JVMs [21]. This is significant, because it operates across many of the Open Systems Interconnection (OSI) layers (from the application, session, network and transport layers). For example, at the enterprise level, *JBOSS* actually provides customized logic in order to extend its own invocation layer and handle bidirectional communications protocols to improve throughput. However, embedded processors provide this directly because they integrate the software and hardware layers within silicon. Figure 2 depicts the progressive devolution of JVM into the physical machine. This diagram is provided to portray the significant benefits of reducing the physical constraints and added complexity encountered by many programmers when dealing with target platforms and OSs. The image on the left shows the JVM supported within traditional computer platforms, while that on the right represents a physical Java processor embedded in silicon.

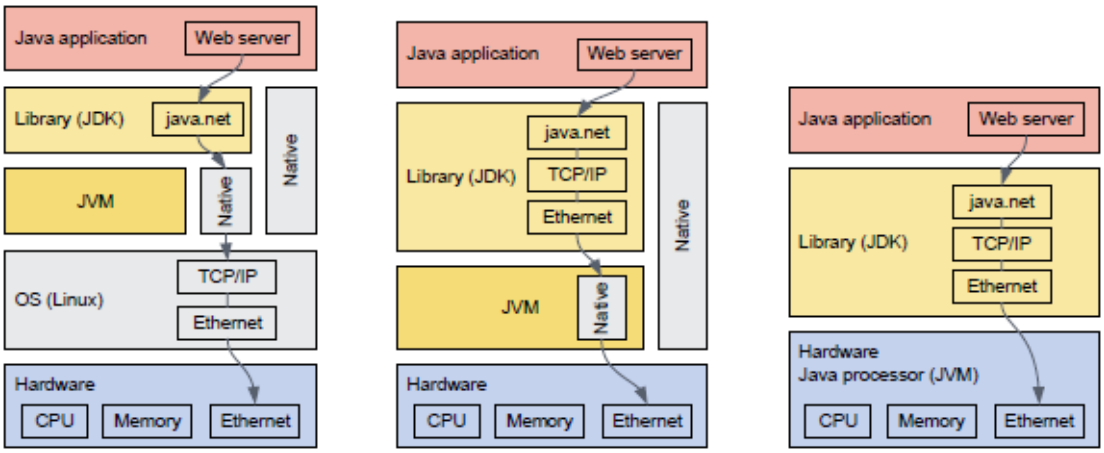

Figure 2: Devolving the JVM in Hardware [22]

To execute Java software on traditional computer platforms, three approaches have been implemented. These include: traditional interpretation, natively compiled execution and a direct execution. It should be noted that Java, like most modern languages, is written using a specific syntax using human readable source code (programme). The programme is then compiled (using Javac) into an intermediate symbolic facsimile in the form of bytecode (normally in one or more libraries called jar files). Given a system that supports Java, the user can execute this application using the OS. The style of execution will depend on how the system was designed, however the three approaches diverge.

Traditional Interpretation: When the bytecode (application) is run using a traditional interpretation method on a Personal Computer (PC), the operation system will

invoke the JVM and pass the bytecode to the JIT compiler to translate the incoming stream into executable binary that is sent to the CPU and processed like any other application.

Native Compilation: Prior to employing JIT compilation, a natively compiled approach was used. Regardless of the approach, the bytecode must be cross-compiled into machine readable binary code (targeted to run on the technology used to host the application (commonly using AMD, Intel and Advanced RISC Machine (ARM) microprocessors instruction sets)).

Direct Execution: The bytecode stream is simply directed to a co-processor that has logic circuits designed to execute bytecode instructions natively (as machine instructions using the same concepts employed by CPUs to execute their binary instruction sets). A Java Processor could be used in three modes. It could be configured as a co-processor, an embedded processor within its own SOC (typically using an FPGA) or designed as a desktop system like the traditional PC. This report focuses on the SOC approach, where the solution is embedded into an Xilinx XC3S500 FPGA, as described in Section 4.

The literature provides significant history about a number of success stories. Each provides enough information to glean the lessons learned. The important lessons are collated in Appendix C. This includes a brief description of the kilo Virtual Machine (C.1), PicoJava (C.2), PicoJavaII (C.3), JEM (C.4), aJile (C.5), and the JavaCard (C.6) followed by a list of other notable research (C.7).

3.5 A Java Optimised Processor within Silicon

The Java Optimized Processor (JOP) provides a processor design that is capable of directly executing Java byte code. It originated from academia and has evolved to provide access to RTSJ using FPGAs. The architecture utilizes a stack based machine design, with a compact set of instructions. The internal architecture relies on a four stage pipe-line design (1+3) which provides efficient processing of Java bytecode. Traditional JVMs currently conduct Complex Instruction Set Computer (CISC) like instructions, where the JOP uses a Reduced Instruction Set Computer (RISC) style approach to improve speed and efficiency[18]. Like its predecessors, the JOP transforms the traditional concept of an Instruction Set Architecture (ISA) which is traditionally supported by an operating system using a hierarchy of calls to a physical machine that directly supports a higher-level language programs [9].

A detailed discussion on JOP is provided in Appendix D. The description is composed of nine sections that include: a description of how the JOP was designed and implemented (D.1), the microcode used (D.2), physical pipelines (D.3), caches (D.4), hardware interrupts (D.5), techniques to process bytecode folding (D.6), garbage collection (D.7), with additional information about multi-core designs using Multi-threaded JVMs (D.8), BlueJEP (D.9) and the CMP (D.10), while D.11 describes how Corba can be used to promote interoperability.

[18] At present the JOP requires bytecode to be preprocessed via an optimizer, which translates the bytecode file into a non-complex set of instructions.

3.6 Conceptual Summary

With respect to this review, a traditional JVM can be summarised as software that is targeted to a specific platform. Such software enables the contents of a predefined file to be parsed, loaded, verified and interpreted at run-time before being executed natively. Java provides developers with the ability to free the user from the desktop by delivering network enabled products to hand held devices or mobile computers (netbooks, ultra laptops and even tablets). Each instantiation of a JVM can be connected to a stream of bytecode. This source file has been previously compiled into a bytecode file that can be piped to a stream to be distributed and executed on supporting heterogeneous platforms. Two major impediments to adopting Java have been its perceived slow speed and lack of ability to deliver enterprise level applications. Most desktop platforms use JIT compilers to achieve near real-time execution to alleviate these issues (approximately 90% of real-time binary code). JIT compilers are embedded into many standard computer systems, many now with clock frequencies exceeding 4 GHZ. Unfortunately the actual throughput of bytecode on these machines is far from predictable due to the OS. For Defence applications, we need real-time systems that provide deterministic assurance. Many JIT compilers prohibit real-time systems, therefore alternatives are required, such as direct execution using RTSJ running on one or more JOP cores.

At present we are often requested to customise designs or *blackbox* implementations, to provide solutions, prior to integration. Customising solutions using FPGAs typically involves a significant amount of effort, resources and expertise. However this approach will generally absorbs the integration risk currently being borne by the Commonwealth. Ideally we desire a seamless method of interoperably exchanging customised subsets of information within acceptable time constraints. Existing research shows that software engineers focus on individual systems to address specific problems, typically at the expense of interoperability, complexity, system constraints and often increased integration risks [23].

Industry and Governments around the world acknowledge the need for standards. They reduce the complexity of describing a need but make contracting and procurement processes easier to verify, while enabling reuse and interoperability. There is no single product or interface that can uniquely solve every problem, but there is a clear need to create a common protocol to assist with the integration and interoperability of C^4ISR assets. A SOC design that embeds a JVM or JOP within an FPGA can have a customised interface containing both analogue and digital circuits that are generated to allow the solution to be connected to almost any equipment in the field. This approach can make it easier for engineers to rapidly leverage from previous designs and efficiently abstract a significant portion of the complexity out of the final solution. This improves time to market and enhances interoperability while retaining a common framework. By embedding one or more JVM cores into an FPGA using VHDL to produce a Silicon solution, it is feasible that higher level languages could deliver enterprise level operations closer to the source. This means that raw data can be pre-processed into meta-data with less skill and streamline the need for off-board computation prior to distribution. This solution would enable engineers to seamlessly adapt existing systems at low cost, in an acceptable and efficient manner, using the highest level of abstraction, with reduced maintenance or obsolescence issues.

4 Demonstrating the Virtual Machine in Silicon

Prior to verifying the principle of using this flexible approach to interfacing legacy equipment in the field, a prototype was physically configured using a Spartan 3 demonstration board. This was achieved by leveraging a collaborative agreement with Prof. Schoeberl from the University of Demark (who provided some documentation and the JOP design libraries). Subsequently a prototype was configured to demonstrate a physical JVM working within a Xilinx XC3S500 FPGA. The success of this experiment required a significant learning curve to ensure the required skills were available. A significant amount of research is also required to ensure the background information is obtained prior to embarking on any synthesis. An initial survey of the domain space revealed a number of commercial products existed. These were based on the original effort by Sun Microsystems. More recent effort by a number of academic endeavours has since been published. The most prolific of these was the JOP. This design ran over 500 times faster than a embedded machine using an interpreted JVM [24]. It also has the highest native clock frequency of those surveyed. Regardless of which design is chosen, each calls on the mastery of several sophisticated tools and a customized environment. The basic process of producing a working platform capable of demonstrating the prototype example is shown in Figure 3. This is followed by a brief introduction to the tools and the development environment used.

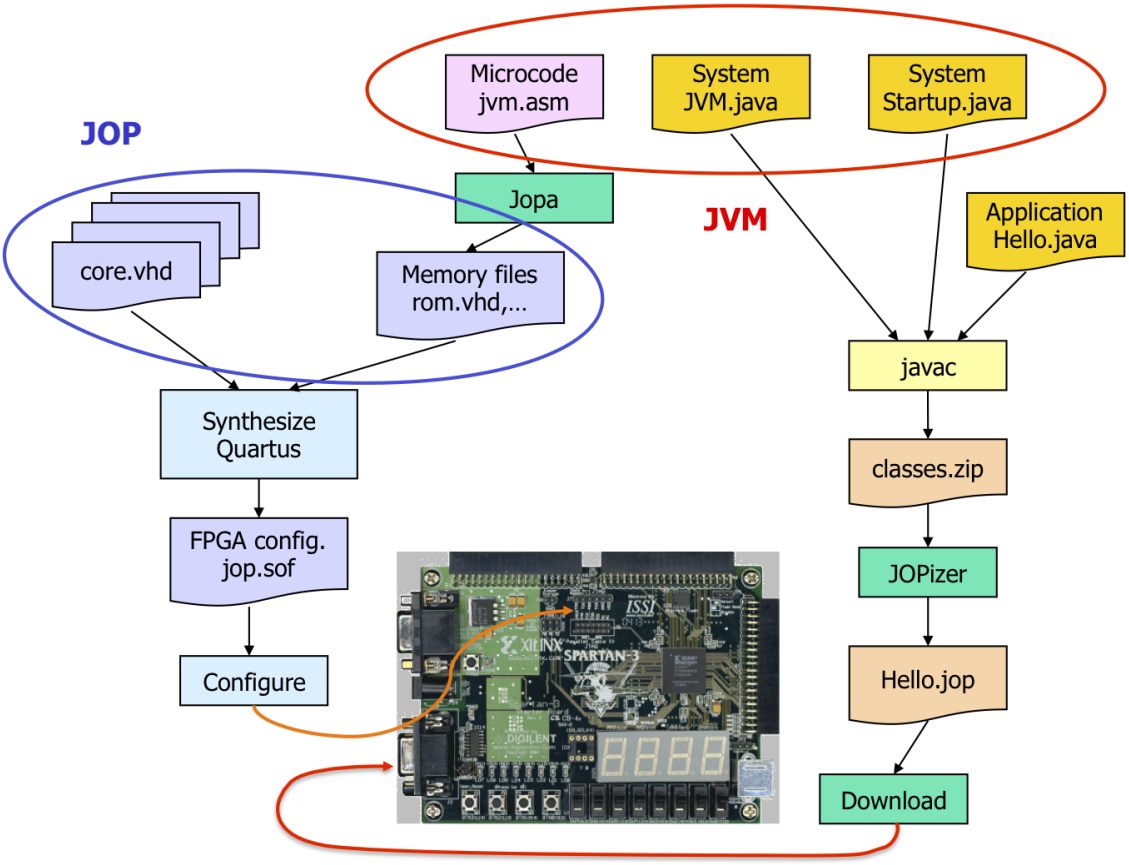

Figure 3: The Concept of using the JOP

4.1 Tools

To complete this project a multi-disciplined team is required as there are a large number of tools required to generate a successful demonstration. Each tool has an inherent learning curve that takes time and significant practice to master. An appropriate environment for each tool needs to be set prior to installation and test. The latter is to verify any issues relate to the *actual* design and NOT the tools. These include:

- cygwin and/or Linux;
- cvs or git;
- gcc and make;
- Java Developers Kit (JDK);
- Eclispe and/or NetBeans;
- VHDL;
- Xilinx ISE and/or Altera's Quartus II.

Alternatives tools can be chosen, but the existing libraries are written in C++, Java and VHDL, while each tool required is only targeted to a limited number of operating systems. Because *ISE* for this work was targeted to Windows, *cygwin* was installed to enable the use of *gcc* on the same platform.

4.2 The Development Environment

There have been numerous commercial attempts at exploiting the direct execution of bytecode in a hardware level JVM. Each had their own reasons, but Sun Microsystems suggested that the 'write once, run anywhere' approach would be verifiable and extremely beneficial because it is secure, well behaved and directly connected to the transport layer[19]. To do this without an operating systems also reduces costs, processing overheads and system complexity. This espouses the concept that an *intelligent client* that is *thin* and easily adapted to the users' needs is now in reach.

Prior to synthesising the JOP the developer must identify the technology being used to host the actual SOC design. Based on the supplied information the appropriate development environment can be chosen and the necessary tools or support libraries installed. In this case a *Xilinx Spartan 3* FPGA chip, mounted on a product evaluation board was obtained (shown in Figure 4). This is supplied with a number of I/O connectors and has several integrated peripherals. These include the display, switches, Light Emitting Diodes (LEDs) and three expansion ports. Some of this I/O is used during the test application.

[19]This concept will be debated by some programmers, because without rigid version controls, even subtle changes can result in the need to patch code.

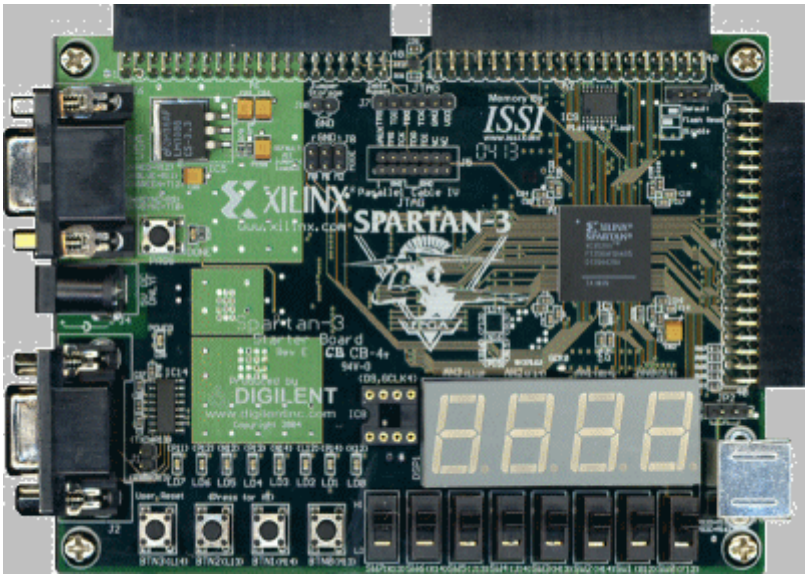

Figure 4: Spartan 3 Board

This board contained the *XC3S500* variant of the chip in an *FT320* ball grid array format. This chip was identified because it contains internal Static Random Access Memory (SRAM) which is used to create on-board stacks. The crucial part of the design is the mapping of the chip resources with the intended I/O. This mapping is done once prior to synthesis, but may require updates with physical design changes. Figure 5 shows the functional schematic of the evaluation board and includes the concept of the chip connectivity.

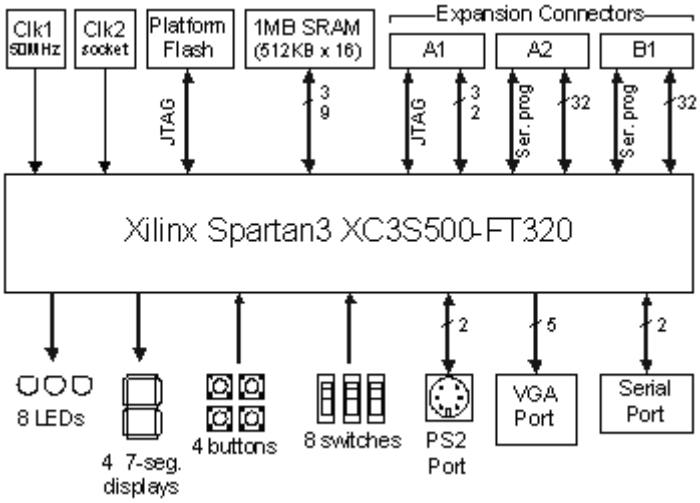

Figure 5: Spartan 3 Functionality

Schoeberl provided a reference handbook on the JOP [25]. He also generously established a "getting started"[20] web page for his group. The web page provides instructions on how to

[20] See http://www.jopwiki.com/Getting_started

download the source Internet Protocol (IP), obtain the tools, instructions on assembling the microcode, compiling an example Java application and getting it to run. This site currently supports designs for the:

- Xilinx Spartan-3,
- ACEX EP1K50C144 JOP core, and
- Cyclone EP1C6/12.

This research also exposed a multi-core design which is discussed as future work.

4.3 Service Based Interfaces and Interoperability

The growth of information, the increasing diversity of information systems and the growing popularity of high-speed network connections continue to challenge enterprise system integration in several aspects [26]:

- Variety of system platforms and programming languages,
- Coexistence of client-server or mainframe oriented system application,
- Lack of a well-defined architecture, and
- Conflicting data formats and semantic definitions.

Over the past two decades a number of data passing methodologies have evolved to improve service-based interfaces and interoperability. These include:

- Distributed Component Object Model (DCOM);
- Distributed Computing Environment (DCE);
- RESTful web framework for Java (REST) (light traffic[21]);
- Simple Object Access Protocol (SOAP) (heavy traffic); and
- Grid or Clustered computing.

There are still situations where Common Object Request Broker Architecture (CORBA) and Data Distribution Service (DDS) provide suitable solutions. These might include cases where:

- Building a distributed system involves multiple programming languages and multiple platforms,
- System entails sending complex data structures where SOAP is not efficient enough,
- There is a significantly high rate of messaging that HTTP cannot support, or
- Legacy applications must be used.

[21] When combined with JavaScript Object Notation (JSON) and Remote Procedure Call (RPC) clusters, REST can be scaled.

4.4 Practical Use-Case

Boeing introduced its *PRiSMj* software to implement the *Ovm*[22] real-time JVM on-board the Scan-Eagle Unmanned Air Vehicle (UAV) platform in 2011 [27]. Figure 6 depicts the Scan Eagle system during the launch phase of flight.

Figure 6: The Boeing Scan Eagle UAV [27]

This is claimed to be Boeing's first attempt to fly a UAV using a real-time Java System [27]. *PRiSMj* was chosen because it supported RTSJ and because of the claim that it provided better performance than using native C++ and presented less issues with portability [28]. This project was sponsored by Defense Advanced Research Project Agency (DARPA) and uses a tertiary thread prioritisation protocol using an event notification mechanism with an update frequency of 1, 5 and 20 Hz. Given the commercial nature of this project, the existing documentation infers it uses Ahead of Time (AOT) compilation to maximize the opportunities for optimization. This resulted in some of the native interfacing being cross compiled (approximately 15,000 lines of C code), however the global application is composed of over 250,000 lines of Java bytecode. Unfortunately there is no real evidence that the *OVM* supports the direct execute bytecode [29].

4.5 Alternative uses in Defence

Defence Science and Technology Group (DST Group) provides technical advice on platform mission system support and operations. There is also significant experience at the enterprise level, especially with platforms like the Airborne Early Warning and Control, which uses The Ace ORB (TAO) operating in CORBA for distributed mission system support. TAO[23] is CORBA 3.0 compliant and interoperability for this framework is al-

[22] From Purdue University, see http://www.ovmj.org/
[23] See http://www.theaceorb.com/

ready supported in Java using *JacORB*[24]. The *JacORB* has been incorporated into *JBoss* to enable the seamless interoperability between Java 2 Enterprise Edition (J2EE) and CORBA enterprise environments. As most web-services are developed in Java, a new architecture for Real-Time Distributed Embedded Systems is proposed. Figure 7 represents this concept.

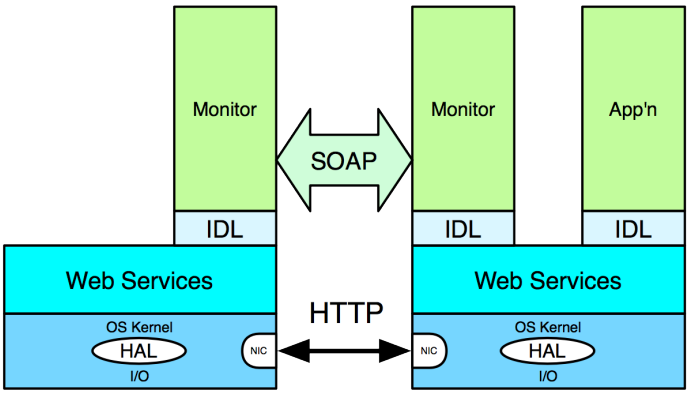

Figure 7: Possible Architecture in Real-Time Distributed Embedded Systems

It is recommended that DST Group conducts more research to advance the concept of interoperability between well established enterprise level systems and the legacy systems currently fielded. At present the Australian Defence Force (ADF) relies on industry to provide customised interfaces to facilitate interaction with legacy equipment. Most of these systems are connected using slow, thin communications portals, therefore the efficiency at both levels needs to be considered. Hence, the goal of this venture is to provide a ubiquitous and flexible interface that enables interoperability for both fixed and mobile assets.

The aim is to test at least one physical implementation using experimentation by adapting an Adaptive Communication Environment (ACE)/TAO interface to allow publisher/subscriber access to sub-systems attributes within a JOP. Information has been gleaned from a subset of the available literature, however that literature strongly suggests that there is a need to continue this exploration through experimentation, verification and testing.

5 Conclusion

This report provides a literature review that shows there is sufficient support for further investigation into using a physical Java Processor in future mission systems of the type currently used in Scan Eagle. There is also sufficient academic interest and opportunities to engage in new partnerships to access the technology required. The Appendices in this document provide a description of the basic concepts required to instantiate a physical Java Machine in an FPGA. These instructions are also supported by experimental results and scenarios about how more advanced applications can be provided. The results clearly

[24] Although a new JacORB (Version 3.8) was released in 2016, the Java implementation is currently using 2.3.1, see http://www.ociweb.com/products/jacorb

indicate that the JOP is a viable design that should be used to champion on-going research into mobile, multi-core, distributed SOC interfaces. This concept is still evolving and more research can be expected before a single standard emerges, however the benefits for both DST Group and Defence are significant.

6 Acknowledgements

The author would like to acknowledge the work done by Martin Schoeberl and his group at the Technical University of Denmark for their ongoing work in developing Java in Silicon expertise and increasing collaboration with a rapidly growing research community.

References

1. Higuera-Toledano, M. T. & Wellings, A. J., eds (2012) *Distributed, Embedded and Real-time Java Systems*, Springer-Verlag, Berlin.

2. Tweedale, J. (1998) *Examining the Relevance of the Java Programming Language*, Honors thesis, Monash University, Melbourne, Vic., Australia.

3. Lund, T., Torralba, A. & Carvajal, R. (2000) The architecture of an fpga-style programmable fuzzy logic controller chip, *Computer Architecture Conference, 2000. ACAC 2000. 5th Australasian* pp. 51–56.

4. Krips, M., Lammert, T. & Kummert, A. (2002) Fpga implementation of a neural network for a real-time hand tracking system, *Electronic Design, Test and Applications, 2002. Proceedings. The First IEEE International Workshop on* pp. 313–317.

5. Sanchez-Solano, S., Senhadji, R., Cabrera, A., Baturone, I., Jimenez, C. J. & Barriga, A. (2002) Prototyping of fuzzy logic-based controllers using standard fpga development boards, *13th IEEE International Workshop on Rapid System Prototyping (RSP'02)* **00**, 25.

6. Tweedale, J. W. (2009) *Autonomous Agent Teaming: Providing Enhanced Communications using Dynamic Components*, PhD thesis, University of South Australia, School of Electrical and Information Engineering, Division of Information Technology, Engineering and Environment, Mawson Lakes, Adelaide, Australia.

7. Gosling, J. & McGilton, H. (1995) *The Java Language Environment: A White Paper*, Technical report, Sun Microsystems, Mountain View.

8. Tweedale, J. & Jain, L. C. (2011) *Embedded Automation in Human-Agent Environment*, Vol. 10 of *Adaptation, Learning, and Optimization*, Springer Berlin, Heidelberg.

9. McGhan, H. & O'Connor, M. (1998) Picojava: a direct execution engine for java bytecode, *Computer* **31**(10), 22 –30.

10. McMillan, W. W. (2009) Java's forgotten forebear, *IEEE Spectrum*.

11. Hayes-Roth, R. (2011) *Hyper-Beings: How InteIligent Organisations Attain Supremacy through Information Superiority*, Booklocker, USA.

12. Scarpino, M. (2008) *Programming the Cell Processor: For Games, Graphics, and Computation*, Prentice Hall, Indiana, USA.

13. Moore, G. E. (1965) Cramming more components onto integrated circuits, *in Electonics*, Vol. 38(8), pp. 1 – 4.

14. Merritt, R. (2015) 5nm test lights litho path, *Electronic Engineering Times, UBM, Santa Monica* pp. 1 – 3.

15. Merritt, R. (2015) 10nm SRAM, 10-core SoC at ISSCC, *Electronic Engineering Times, UBM, Santa Monica* pp. 1 – 4.

16. Demerjian, C. (2010) Global foundries goes to euv litho at 15nm, *SemiAccurate News, Minneapolis, MN*.

17. Krazit, T. (2006) Intel pledges 80 cores in five years, *CNET News, CBS Interactive, Sydney Australia*.

18. Fulton, S. M. (2007) Intel to show off 1 TFlop, 80-core CPU, *BaetaNews, Chicago*.

19. Gilder, G. (1993) *Metcalfes Law and Legacy*, Technical report, Forbes.

20. Rubenstein, R. (1996) Sun plans feast of Java processors, *Electronics Weekly, Reed, Haywards Heath* (1754), 1–8.

21. Jach, A. (1996) Tomorrow's CPU's, *Byte, McGraw-Hill, London* **11**, 76.

22. Schoeberl, M., Korsholm, S., Thalinger, C. & Ravn, A. P. (2008) Hardware objects for java, *in Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, IEEE Computer Society, pp. 445–452.

23. Bray, D. A., ed. (2007) *Information Systems*, University of Georgia, USA.

24. Pitter, C. & Schoeberl, M. (2008) Performance evaluation of a java chip-multiprocessor, *in Proceedings of the 3rd IEEE Symposium on Industrial Embedded Systems (SIES 2008)*, IEEE, pp. 34–42.

25. Schoeberl, M. (2009) *JOP Reference Handbook: Building Embedded Systems with a Java Processor*, CreateSpace.

26. Gong, Y. (2003) *CORBA Application in Real-Time Distributed Embedded Systems*, Ece 8990 survey report, Mississippi State University, USA.

27. Baker, J., Armbruster, A., Flack, C., Pizlo, F., Prochazka, M., Vitek, J., Holmes, D., Armbruster, A. & Pla, E. (2006) A real-time java virtual machine for avionics, an experience report, *in 12th IEEE RealTime and Embedded Technology and Applications Symposium RTAS*, ACM Press, pp. 384 – 396.

28. Henties, T., Hunt, J. J., Locke, D., Nilsen, K., Schoeberl, M. & Vitek, J. (2009) Java for safety-critical applications, 2nd international workshop on the certification of safety-critical software controlled systems, *Science* pp. 1–11.

29. Flack, C., Hosking, T. & Vitek, J. (2003) *Idioms in Ovm*, Technical Report CSD-TR-03-017, Purdue University, Lafayette, IN, USA.

30. Doyle, M. (1996) Proposing a standard web api, *Dr. Dobb's Journal, Miller Freeman* **21(2)**, 18–26.

31. Singleton, A. (1996) Wired on the web, *Byte, McGraw-Hill, London* **21(1)**, 77–80.

32. Lemay, L. & Perkins, C. (1996) *Teach Yourself Java in 21 Days*, Sams, Indianapolis.

33. Arnold, K., Gosling, J. & Holmes, D. (2005) *The Java Programming Language*, The Java Series, 4th edn, Sun Microsystems.

34. White, G. (1996) Java command-line arguments, *Dr. Dobb's Journal, Miller Freeman* **21(2)**, 58–61.

35. Joy, B. (2000) *J2ME Building Blocks for Mobile Devices*, Sun Microsystems, Palo Alto, CA.

36. Zhioua, S. (2008) *A Dynamic Compiler for an Embedded Java Virtual Machine*, VDM Verlag, Saarbrücken, Germany.

37. Koopman, P. J. (1989) *Stack Computers: the new wave*, Ellis Horwood.

38. Miller, A. (2008) What to expect in Java SE 7, *Java World* **12**, 1–8.

39. Lindholm, T. & Yellin, F. (1999) *The Java Virtual Machine Specification*, 2nd edn, Addison-Wesley, Reading, MA.

40. Null, L. & Labour, J. (2006) *The Essentials of Computer Organization and Architecture*, Jones and Bartlett, Sunbury, MA.

41. Haggar, P. (2001) Understadning bytecode makes you a better programmer, *IBM Developer Works, Triangle Park, North Carolina* pp. 1 – 7.

42. Arhipov, A. (2012) Mastering java bytecode at the core of the JVM, boston, ma, *Zero Turnaround,* pp. 1 – 46.

43. Rauh, S. (2015) A quick guide to writing bytecode with ASM, wordpress, ny, *Concepts of programming languages,*.

44. Collberg, C., M., M. M. & Huntwork, A. (2003) Sandmark: A tool for software protection research, *IEEE Magazine of Security and Privacy* **1(4)**, 40 – 49.

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

46. Gruian, F. & Westmijze, M. (2007) Bluejamm: A bluespec embedded java architecture with memory management, *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on* pp. 459–466.

47. Henriksson, R. (1998) *Scheduling Garbage Collection in Embedded Systems*, PhD thesis, Department of Computer Science, Lund University, Sweden.

48. Ungar, D. (1985) Generation scavenging: a nondisruptive high performance storage reclamation algorithm.

49. Robertz, S. G., Henriksson, R., Nilsson, K., Blomdell, A. & Tarasov, I. (2007) Using real-time java for industrial robot control, *in Proceedings of the 5th International Workshop on Java technologies for real-time and embedded systems*, JTRES '07, ACM, New York, NY, USA, pp. 104–110.

50. Masson, D. & Midonnet, S. (2008) Rtsj extensions: Event manager and feasibility analyzer, ACM Press.

51. Sun Microsystems (1997) *Java Processor Core*, Data Sheet 805-2990-01, Sun, Palo Alto, CA.

52. O'Connor, J. & Tremblay, M. (1997) picojava-i: the java virtual machine in hardware, *Micro, IEEE* **17**(2), 45 –53.

53. Dey, S., Panigrahi, D., Chen, L., Taylor, C., Sekar, K. & Sanchez, P. (2000) Using a soft core in a soc design: experiences with picojava, *Design Test of Computers, IEEE* **17**(3), 60 –71.

54. Sun Microsystems (1999) *picoJava-II Programmer's Reference Manual*, Data Sheet 805-2800-06, Sun, Palo Alto, CA.

55. Joy, B. (1999) *picoJava-II Microarchitecture Guide*, Sun Microsystems, Palo Alto, CA.

56. Joy, B. (1999) *picoJava-II Programmer's Reference Manual*, Sun Microsystems, Palo Alto, CA.

57. Aoki, T. (2001) Proceedings of the java virtual machine research and technology symposium, *USENIX Association* pp. 1–12.

58. Peng, V. (1999) *JTAG Programmer Guide*, Technical report, Xilinx, Inc., California.

59. Kuwahara, C., Pomeroy, B. & Ikeda, S. (2000) Fijitsu announces java solution for fast-growing network application market, Press Release.

60. Aoki, T. (2001) On the software virtual machine for the real hardware stack machine, *in Proceedings of the Java Virtual Machine Research and Technology Symposium*, USENIX Association, Monterey, CA, pp. 1 – 12.

61. Puffitsch, W. & Schoeberl, M. (2007) picojava-ii in an fpga, *in Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, ACM Press, pp. 213–221.

62. Sun Microsystems (1999) *picoJava-II Microarchitecture Guide*, Data Sheet 960-1160-11, Sun, Palo Alto, CA.

63. Kuwahara, C., Pomeroy, B. & Ikeda, S. (2008) Open source software: opportunities for cost reduction, Fact Sheet.

64. Horowitz, S. (2009) ajile's latest system-on-chip processor brings low-cost, low-power java jolt to m2m, industrial, and remote real-time network applications, *New Java^{TM} Technology SoC Simplifies Smart Infrastructure, Bloomberg, San Jose, CA* pp. 1 – 2.

65. Ploog, H., Kraudelt, R., Barrow, N., Rauchui, T., Golatowski, F. & Timmermann, D. (1999) A two step approach in the development of a java silicon machine (jsm) for small embedded systems, *Workshop on Hardware Support for Objects and Microarchitectures for Java, Austin, Texas* pp. 1 – 5.

66. Joy, B. (2009) *Java Card Specification 3.0.1*, Technical report, Sun Microsystems.

67. Golatowski, F., Ploog, H., Bannow, N. & Timmermann, D. (2002) Jsm: A small java processor core for smart cards and embedded systems, *in International Conference on Architecture of Computing Systems*, Trends in Network and Pervassive Computing - Java in Embedded Systems, pp. 135 – 140.

68. Kreuzinger, J., Marston, R., Ungere, T., Brinkschulte, U. & Krakowski, C. (1999) The komodo project: Thread-based event handling supported by a multithreaded java microcontroller, *25th EUROMICRO Conference, Milano, Italy* pp. 122–128.

69. Ito, S., Carro, L. & Jacobi, R. (2001) Making java work for microcontroller applications, *Design Test of Computers, IEEE* **18**(5), 100 –110.

70. Schoeberl, M. (2008) A java processor architecture for embedded real-time systems, *Journal of Systems Architecture* **54/1–2**, 265–286.

71. Zabel, M. & Spallek, R. G. (2010) Application requirements and efficiency of embedded java bytecode multi-cores, *in Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, ACM, New York, NY, USA, pp. 46–52.

72. YiYu, T., Man, L. K. & Fong, A. (2006) A performance analysis of an object-oriented processor, *in Information Technology: New Generations, 2006. ITNG 2006. Third International Conference on*, pp. 690 –694.

73. Distefano, D. & Parkinson J, M. J. (2008) jstar: towards practical verification for java, *SIGPLAN Not.* **43**, 213–226.

74. Hardin, D. (2001) Crafting a java virtual machine in silicon, *Instrumentation Measurement Magazine, IEEE* **4(1)**, 54 – 56.

75. Beck, A. C. S. & Carro, L. (2003) Low power java processor for embedded applications, *in VLSI-SOC*, Technische Universität Darmstadt, Insitute of Microelectronic Systems, pp. 239–255.

76. Kreuzinger, J., Brinkschulte, U., Pfeffer, M., Uhrig, S. & Ungerer, T. (2003) Real-time event-handling and scheduling on a multithreaded java microcontroller.

77. Schoeberl, M. (2008) *JOP: A Java Optimized Processor for Embedded Real-Time Systems*, VDM Verlag Dr. Müller.

78. Schoeberl, M. (2006) A time predictable java processor, *in Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, pp. 800–805.

79. Ton, L.-R., Chang, L.-C., Kao, M.-F., Tseng, H.-M., Shang, S.-S., Ma, R.-L., Wang, D.-C. & Chung, C.-P. (1997) Instruction folding in java processor, *in Parallel and Distributed Systems, 1997. Proceedings., 1997 International Conference on*, pp. 138 –143.

80. Schoeberl, M. (2005) Evaluation of a java processor, *in Tagungsband Austrochip 2005*, pp. 127–134.

81. Schoeberl, M., Huber, B., Binder, W., Puffitsch, W. & Villazon, A. (2010) *Object Cache Evaluation*, Technical report.

82. Schoeberl, M. (2004) A time predictable instruction cache for a java processor, *in On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, Vol. 3292, Springer, pp. 371–382.

83. Schoeberl, M. (2011) A time-predictable object cache, *in Proceedings of the 14th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2011)*, IEEE Computer Society, pp. 99–105.

84. Schoeberl, M. & Hilber, P. (2010) Design and implementation of real-time transactional memory, *in Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL 2010)*, IEEE Computer Society, pp. 279–284.

85. Schoeberl, M. & Pedersen, R. (2006) Wcet analysis for a java processor, *in Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, ACM Press, pp. 202–211.

86. Huber, B. & Schoeberl, M. (2009) Comparison of implicit path enumeration and model checking based wcet analysis, *in Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, OCG, pp. 23–34.

87. Huber, B. & Schoeberl, M. (2008) *Comparison of ILP and Model Checking based WCET Analysis*, Technical Report 72/2008.

88. Gruian, F. & Westmijze, M. (2007) Bluejep: a flexible and high-performance java embedded processor, *in Proceedings of the 5th International Workshop on Java technologies for real-time and embedded systems*, JTRES '07, ACM, New York, NY, USA, pp. 222–229.

89. Pedersen, R. & Schoeberl, M. (2006) Exact roots for a real-time garbage collector, *in Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, ACM Press, pp. 77–84.

90. Schoeberl, M. & Puffitsch, W. (2010) Nonblocking real-time garbage collection, *ACM Trans. Embed. Comput. Syst.* **10**(1), 6:1–28.

91. Schoeberl, M. (2010) Scheduling of hard real-time garbage collection, *Real-Time Systems* **45**(3), 176–213.

92. Schoeberl, M. (2006) Real-time garbage collection for java, *in Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, IEEE, pp. 424–432.

93. Mun, C. K., Krishna. Rao, G., Lam, H.-S., Eswaran, C. & Phon-Amnuaisuk, S. (2007) Performance optimization of java virtual machine on dual-core technology for web services - a study, *in Advanced Communication Technology, The 9th International Conference on*, Vol. 1, pp. 567 –570.

94. Chung., C.-M. & Kim., S.-D. (1998) A dualthreaded java processor for java multithreading, *in Parallel and Distributed Systems, 1998. Proceedings., 1998 International Conference on*, pp. 693 –700.

95. Pitter, C. & Schoeberl, M. (2007) Towards a java multiprocessor, *in Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, ACM Press, pp. 144–151.

96. Schoeberl, M., Puschner, P. & Kirner, R. (2009) A single-path chip-multiprocessor system, *in Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*, number LNCS 5860, Springer, pp. 47–57.

97. Schoeberl, M. (2008) Application experiences with a real-time java processor, *in Proceedings of the 17th IFAC World Congress*, pp. 9320–9325.

98. Schoeberl, M. (2010) Time-predictable chip-multiprocessor design, *in Forty Fourth Asilomar Conference on Signals, Systems and Computers (ASILOMAR),*, IEEE Signal Processing Society, pp. 2116 – 2120.

99. OMG (2004) *Real-time CORBA*, Specification, version 1.2, Object Management Group, Inc., Needham, MA.

Appendix A The Java Programming Language

Distributed programming is not new, however Java delivered dynamic interaction in a heterogeneous environment facilitated across the Internet [7]. This is achieved by executing Java byte code (compiled Java) on a purpose built virtual machine. The focus of this report is highlighting the linkages required to successfully embed an operational Java Virtual Machine (JVM) in a Field Programmable Grid or Gate Array (FPGA). Currently the JVM uses software libraries to provide a small, reliable, portable, distributed, real-time operating environment that runs seamlessly on many Operating Systems (OSs) and computer architectures. The Java programming language has syntax and functionality that has been influenced by techniques presented in existing languages that included C++, Eiffel, SmallTalk, Objective C, and Cedar/Mesa. Like most modern languages, each implementation is supplied with a predefined set of libraries that are supported by the developers to ensure the uniform implementation of the language across all platforms. Originally Sun Microsystems provided support for Unix, Macintosh and Windows NT/95 platforms and now supports most other platforms. Sun originally championed Java by licensing to Apple, IBM and Microsoft, who all distributed the JVM with their OSs. Access to Java within HyperText Markup Language (HTML) documents was also provided by Doyle and his team in 1993 (using *Applets*[25]) [30]. In some ways, this concept revolutionised the way humans interact with computers and information. Today JavaScript, Flash and HTML5 are more frequently used as underlying code for Graphical User Interfaces (GUIs).

In 1996 Singleton reported that James Gosling actually started developing Java in 1991 while working on the Network extensible Window System (NeWS) [31][26]. It was quickly retitled as *Oak*[27] in an attempt to expand into the Personal Digital Assistant (PDA) market; however, eventually drifted until it became Java in 1994. The Java language was promoted by Gosling in a white paper [7], and supported by Arnold with a full programming reference guide [33]. Sun Microsystems commercially released Java as a product by introducing the Java Developers Kit (JDK) for a targeted number of computer architectures. Java Intelligent Network Interface (JINI) introduced to promote the use of Java on mobile devices, and it soon became a significant framework, because it removed the need for physical peripherals or interfaces using networked services. The Java language is designed to operate with objects of a number of predefined types, although the JVM only supports primitive types natively. Objects are constructed as a structured element that is managed using classes that where boundary aligned in 8-bytes increments in memory (unused bit or bytes are padded). The primitive types and their native representation are also consistent with most programming languages.

Java is promoted as a less complex implementation of C++. Its syntax has the familiarity of C++, but is less confusing because the programmer does not need to deal with pointers, pre-processing, multiple inheritance, label or automatic coercion [34][28]. Some performance

[25]These Applets initially consisted of animated characters, sprites and simple games; however, devotees and industry are now producing serious Java applications that run natively.

[26]The original project was called *Green*, which was aimed at implementing a heterogeneous network of electronic consumer products.

[27]This variant was supported to demonstrate the concept of the 'set-top-box' [32].

[28]It should be noted that you can use 'Goto-like' labels and implicit coercion without the loss of precision (when promoting types), although the more recent introduction of 'auto boxing' can enable the programmer to introduce some very nasty bugs.

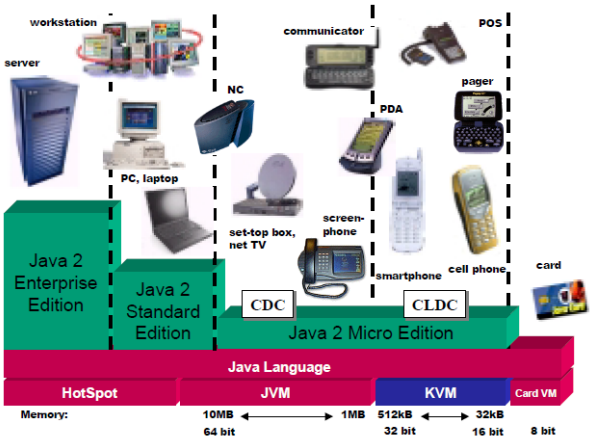

Figure A1: Java2 JVM Targets [35, 36]

was compromised to deliver type-safe features, that include references, index safe arrays and automatic garbage collection. This approach provided a fresh, cleaner, safer programming environment for beginners and professionals. Presently programmers are forced to compile their code (classes) prior to execution; however, modern development tools reveal the possibility of a fully interpreted environments similar to Basic by utilizing the power of compiled languages [2]. Since its release, Java has undergone several significant upgrades (1.2, 1.4, 1.5, 1.6, 1.7 and now 1.8).

This evolution stagnated for several years after Oracle acquired Sun Microsystems[29]. During this period a number of alternative JVM stack-based languages like Closure, Groovy, Scala and Gershwin emerged [37][30]. Another set of commentators from IBM and Oracle believe that the next generation of Java will *fork* into something with a complete new direction (retaining its backwards compatibility)[31]. Regardless of direction, there has been significant effort on a variety of projects focused on retaining the JVM, including more than 60,800 links on *sourceforge*[32]. This is far more than any other language hosted on this site. Java libraries exist that now support, interpret or emulate the syntax of many common languages. With recent releases, Java has been improved to include concurrent processes and software developers need to consider processing objects in parallel. Java was primarily used for teaching in universities, however has extended its reach to enthusiasts, professionals and even enterprise application development.

A.1 The Current Mainstream Release - Java SE8

Java is still evolving with the latest release set at SE8u144 (September 2017). Java classes are still used to define templates that can instantiate objects, complete with abstracted object data types, properties and methods. Object Oriented Programming (OOP) programmers use classes to encapsulate the state and behaviour of objects within a problem

[29] See http://www.datamation.com/entdev/article.php/3862566/Whats-Javas-Future.htm
[30] See http://www.ece.cmu.edu/~koopman/projects.html#stack
[31] See http://www.sdtimes.com/content/article.aspx?ArticleID=34820&print=true
[32] See http://sourceforge.net/directory/os:windows/?q=java

space. Software is increasingly being written using inheritance and event messaging to support a systems approach to problem solving (often using a distributed paradigm). Agents are being employed to make decisions and support autonomous behaviour. Each class would typically run as an independent thread with the ability to communicate with other classes or agents using Simple Object Access Protocol (SOAP), RESTful web framework for Java (REST) and even JavaScript Object Notation (JSON). The Agent class was first introduced in version SE5 and further refined in version SE6. The Java Management eXtensions (JMX) provided the ability to *analyse* or *test* applications with agents or even applications running in separate JVMs. Java SE7 was released in 2007 and it supports Portable Operating System Interface (POSIX) threads with a richer set of periodic and episodic unit calculations. Java SE8 was released in 2014 with 'Lambda' expressions, collections, compact profiles, improved security, time/date internationalisation, improved tools and Java FX. Table A1 lists a number of notable improvements previously submitted by the Java Community Process (JCP)[33]. Java Specification Requestss (JSRs) are routinely submitted for industry concurrence (with approved requests being issued as part of interim updates) [38]. For a complete listing of current JSR activities, consult the Community Development of Java Technology Specifications page[34].

As suggested, Java evolves based on a community sponsored improvement process. Improvements are frequently highlighted in forums and discussion boards. Several requests reviewed while researching this report include: reducing the need for boilerplate code, better interoperability between classes running on independent JVMs in the same memory space and improved reflected properties using optionally annotated meta-data.

Simplified Boilerplate Code: The *Facade pattern* is commonly used in Java to associate components in a single GUI. Each component is progressively constructed prior to being aggregated into a compounded control. It has been suggested that default implementations for components be accepted for scroll bars, spin controls, mouse, keyboard and move events to simply coding. Embedded components would essentially be abstracted through reflective properties and each object (also called a 'bean') or component could be instantiated using a single call. Programmers should only be called upon to override a default API call when a customized implementation is required. For instance a button generally always requires an event handler. Most GUI programmers also include mouse controls (through click events). Programmers even extend their designs to include keyboard short-cuts (key events) and increasingly some even add gesture controls (touch events). It would be feasible to pass attributes as options to include icons, actions and default values during construction without the additional effort required to build individual components and chaining the object references. If this were the default condition, many aspiring programmers would achieve a more meaningful contribution with less complexity and frustration.

[33]The JCP was established in 1998. This community acts a formalized mechanism that allows interested parties to develop standard technical specifications for consideration in future Java releases. Proposed changes are submitted and championed in an open source arrangement with community members.

[34]See https://jcp.org/en/jsr/all

[39]This will provide methods to manage Episodic, Scientific, Empirical and System time measurement.

[40]This functionality is being progressively supported in Java FX 3D.

[41]See above.

Table A1: Notable JSR Activities

JSR	Description
109	Web-services 1.3. This specification defines the programming model and runtime architecture for implementing web services in Java.
166	Provides *fork* and *join* processes to support concurrency. It will also provide a set of work queues to support *work stealing* to manage idle workers.
203	Extends the work started in JDK 1.4 and will improve *File* and *Directory* permissions.
255	Provide JMX updates to enable the *federation* of remote services.
270	Use components in JTabbedPane.
286	Portlets 2.0. plans to align with Java 2 Enterprise Edition (J2EE) 1.4, integrate other new JSRs relevant for the portlet, and align with the Web Services for Remote Portlets (WSRP) specification V-2.0.
292	Culminated discussion on *tail call optimization* and *interface injection*, however may only see **invokedynamic** included.
294	This was raised to improved modularity and namespaces that became project *jigsaw* when the OSGi Alliance joined the project.
295	Is targeted to improve Swing and Bean binding/validation.
296	Use SwingWorker with JFC & Swing.
299	Better Dependency management similar to that experienced with *Marven*, JBOS Seam, Enterprise Java Beans (EJB) and *OSGi*.
310	Provides better *Date* and *Time* dimensioning in an attempt to provide *socialized* labels which will be great for managing behaviour[39].
330	Services Framework. This specification will define a high level, lightweight services and management framework Application Program Interfaces (APIs) that will provide Java 2 Micro Edition (J2ME) based devices the ability to manage long running applications and services.
366	Expand the multi-threaded and multi-core capabilities within the Fork/Join Framework.
367	JSON API binding layer (metadata & runtime) for converting Java objects to/from JSON messages.
371	This JSR is to develop Model-View-Controller (MVC) 1.0, a model-view-controller specification for Java EE.
912	Version 1.3 of the Java 3D API[40].
926	Maintenance of the Java 3D specification[41].
927	The maintenance of the Java TV specification.

The concepts associated with reuse and flexibility are promoted using interfaces, wrappers and redirection.

Improved Interoperability: Traditionally Object Oriented Programming Languages

(OOPLs) restrict visibility of data or variables using a number of scoping techniques. Many use inheritance to encapsulate or extend object types and behaviour which can necessarily bloat code or make it a challenge to read. The alternative is to use predefined redirection statements (One example is using the *instanceof* call to pass messages within the application). This is based on complex arrangements that connect one or more sockets and queues. The manner in which the programmer addresses the message flow can block execution. For instance a post and wait call would block the programme until a return was provided, whereas a callback function (like that used in a model, view, control widget) would allow the programme to continue and revisit to process the information when the response is signalled. Agents and agent systems benefit from the latter when they are organised to function as independent applications (often in their own JVM). To achieve this in Java, developers can use queue events, but this also compromises asynchronous designs. The preferred alternative is to use a *SwingWorker* function call an independent thread to process the function in the background. This approach enables designers to create independent *Property Change Listeners* as thread that subscribe to components embedded into a GUI design.

Annotated Meta-data: Annotations[42] were introduced to Java in version 5.0 and improved/extended in version 6 to facilitate the use of *meta-data*[43]. Annotations can be used to provide: information to the compiler, conduct both 'compiler-time' and 'deployment-activity' processing or to enable run-time processing. When Annotation are declared they can be applied to the class, specified fields, methods or other program element. The annotation processors generate macro-like code that is automatically completed at build-time[44]. Originally, Java programs were 'decorated' with annotations indicating which methods were remotely accessible. This provided clarity when using Java API for XML-based RPC (JAX-RPC) web service and when relying on Reflection. Annotations are also used to provide *deployment descriptor* for Enterprise Java Beans (EJBs) in order to reduce complexity and enforce its strict syntax requirements. Annotations have proven popular in Java and many tutorials now exist, therefore it will not be the subject of discussion in this report.

The trend in simulation applications is to separate the scenario data from the business logic. Traditionally the business logic is programmed to represent both the environment structure and its behaviour. This concept needs to be further extended to simplify an implementation that supports a dynamic context. The new model should retain the business logic that supports common functionality (model kernel), but separate the context and behaviour. A XML Object Model (XOM) file provides the mechanisms required to encode the behaviour of the changing context to implement patterns[45] using the Java Context and Dependency Injection (CDI) framework in enterprise platforms[46] to *weld* components

[42] JSR 175 - A Metadata Facility for the Java™ Programming Language.

[43] @Annotations were originally implemented to enable developers to introduce *tags* or their own preprocesses, similar to that in other languages.

[44] This is sometimes called boilerplate code and could include support for strict syntax encoding, the generation of Extensible Markup Language (XML) files or even documentation (like *javadoc*). Care should be taken to ensure it doesn't result in confusion or unintentionally pollute the programs semantics.

[45] Java makes use of the *java.util.Observable* patterns.

[46] See JSR-299 for further detail.

using the Java Message Service (JMS) within the base environment and each scenario instantiated.

Existing simulation frameworks are more complex than necessary because they need to address all possible events[47] for any given entity that interacts with the environment. *Behaviour*, *desire* and *intent* are generally infused with the business logic. Alternatively the implementation logic inherits a communication model, such as Knowledge Query Manipulation Language (KQML), Access Control List (ACL) or Foundation of Intelligent Physical Agents (FIPA). SOAP is emerging as the model of choice for distributed communications in Java.

[47] Events contain objects or messages that are used when a software component wants to notify a state of change to another component within any package, thread or application.

Appendix B A JVM in Silicon

There have been a number of Virtual Machines (VMs) designed to support Java over the past two decades. The Sun Microsystems originally offered the concept based on three variants. They took the form of the: picoJava, microJava and ultraJava processors. Sun Microsystems' picoJava architecture was targeted at a wide range of devices from small, hand held appliances to desktop network computers. The *picoJava* chip contains the core architecture of the Java Virtual Machine (JVM) and can be built as physical machine. The *microJava* chip includes the picoJava core plus memory, I/O and other control functions and was targeted towards controllers, network-based devices and consumer products. The *ultraJava* chip was modelled after the UltraSPARC workstation and targeted at desktop use and incorporates some of Sun Microsystems' 3D graphics processing.

B.1 The Java Virtual Machine

Prior to the physical manifestation of a machine that was capable of processing Java bytecode natively, researchers attempted to simulate and verify their ideas. Lindholm and Yellin labelled the outcome as the JVM [39]. The Bytecode instruction set was defined using one byte opcode (8-bits) followed by zero or more operand bytes [40]. The JVM was a software interpreter that abstracted the complexity of the computing machine. This relieved the programmer of many of the low-level constructs associated to interfacing with the platform hardware and Input/Output (I/O). The language focused on processing object typed software using network concepts [39]. It does not assume any particular implementation, technology, operating system or host and processes *class* files constructed from symbolic code with embedded ancillary information compiled from the Java source code [7, 33]. The initial conceptual representation of a JVM is shown in Figure B1.

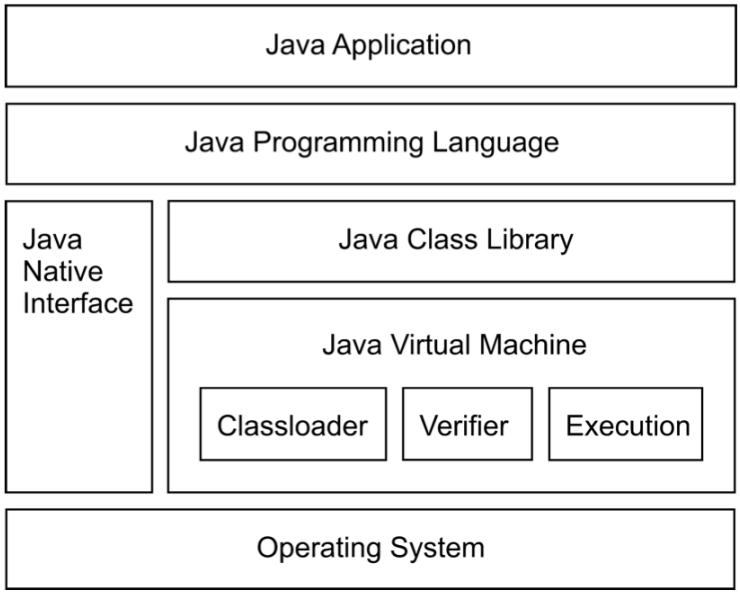

Figure B1: Concept surrounding the JVM

B.2 Java Bytecode

When Java is installed on a Personal Computer (PC), compilation of source code produces a class library that contains bytecode. Execution typically requires greater than 45 kBytes of Read Only Memory (ROM) (containing linked libraries) and a large amount of Random Access Memory (RAM). Enterprise level implementations require hundreds of kBytes to store more complex libraries (typically as jar files). Unfortunately smart phones, Personal Digital Assistants (PDAs) and mobile devices normally have limited resources. Early use of Java on set-top boxes and primitive web-browsers used Java processors to implement dynamic translation. More recent releases have proven to be more capable in Connected Device Configuration (CDC) applications. However, with the increased use of System on Chip (SOC) devices, the scope for native Java Processors has re-surfaced.

There are theoretically 256 bytecodes internally supported by the current JVM, however only 201 are defined. Two of these instructions are reserved as software traps and a third for debugging. The Java Optimized Processor (JOP) defines 226 variable length bytecode instructions. The additional instructions are mapped to replace bytecode engaged in machine level activities, such as *parsing, loading* and *verifying* classes. Figure B2 displays the frequency distribution of JOP bytecode instructions.

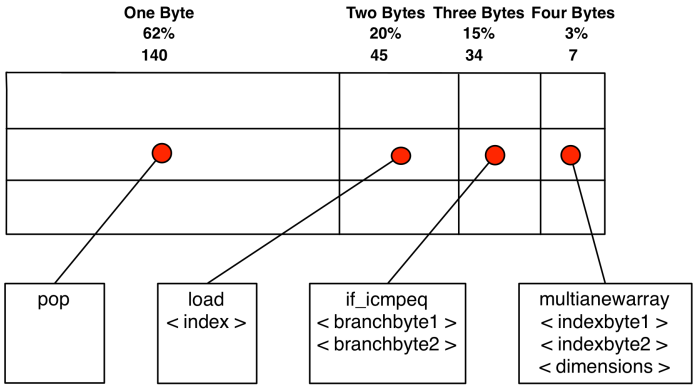

Figure B2: Java Bytecode Distribution

Each column depicts the percentage of opcodes based on their length in bytes, where 62% have one byte, 20% two bytes, 15% three bytes and only 3% have four bytes). This indicates that almost two thirds have no operand and will operate natively using a single instruction cycle, while over 80% of the bytecode will run using in-line data. To take advantage of this fact, the JOP maps 54 of the incoming bytecodes into internal 10-bit hardware instructions (where the additional bits are used to populate the *nxt* and *opd* flags within the processor). At the time of writing, 43 of the 201 different bytecodes are also implemented using single byte microcode machine instructions, 93 with multi-byte internal microcode sequences and the remaining 40 bytecodes are implemented in accordance with the JVM specification.

Figure B3 displays two columns. On the left is a Java programme that adds two variables. On the right is the equivalent compiled bytecode. This example uses a simple stack operation and efficiency should only be calculated by reviewing the bytecode. Using a print statement to visualise the results would employ slow console operations and therefore

benchmarking should only be conducted using a JOP deployed to an Field Programmable Grid or Gate Array (FPGA).

```
public class X {

   public static void
   main(String[] args) {
        add(1, 2);
   }

   public static int
   add(int a, int b) {
        return a+b;
   }
}
```

```
public static void main
   (java.lang.String[]);
  Code:
   0:   iconst_1
   1:   iconst_2
   //Method add:(II)I
   2:   invokestatic    #2;
   5:   pop
   6:   return

public static int add
   (int,int);
  Code:
   0:   iload_0
   1:   iload_1
   2:   iadd
   3:   ireturn
```

Figure B3: Java Bytecode Example

Using the same concepts associated with programming in assembly code and stack operations, a knowledgeable programmer could pack bytecode by hand to produce very tight code (without the need to define, instantiate and manipulate variables or classes). More details can be sourced from the literature. Examples include: Haggar [41], Schoeberl [25], Arhipov [42] and Rauh [43]. Other techniques can be applied to stream line existing code or improve execution using physical hardware (for instance, by introducing a method stack or dynamic access to the stack). After analysing 801,117 randomly sampled methods, sourced from jar-files found in community projects on-line, 73% were found to contain less than nine subroutines [44]. Schoberl introduced a method stack to manage frequently used subroutines. This increases cache coherence and minimises the use of padding instructions and the need to flush the programme cache or use slower I/O [45].

Listing 1 illustrates how a complex print statement in Java (a compounded set of statements to concatenated hard coded 'Text' with a 'Sring' variable) is generally less efficient than a more deliberate example shown in Listing 2. The compiled bytecode of both examples is displayed before its original Java source listing (boxed). The curious reader could use *javap* to generate the same result.

```
getstatic       #3; // Field System.out:Ljava/io/PrintStream;
new             #4; // class StringBuffer
dup
invokespecial   #5; // StringBuffer."<init>":()V
ldc         #6; // String Result =
invokevirtual   #7; // StringBuffer.append:(LString;)LStringBuffer
iload_1
invokevirtual   #8; // StringBuffer.append:(I)LStringBuffer;
invokevirtual   #9; // StringBuffer.toString:()LString;
invokevirtual   #10;// PrintStream.println:(LString;)V
```

```
1    System.out.println(''Running Total ='' + currentValue );
```

Listing 1: Complex Print Statement

```
getstatic       #3; //Field System.out:Ljava/io/PrintStream;
ldc             #4; //String Result =
invokevirtual   #5; //Method PrintStream.print:(LString;)V
getstatic       #3; //Field System.out:LPrintStream;
iload_1
invokevirtual   #6; //Method PrintStream.println:(I)V
```

```
1 System.out.print(''Running Total ='');
2 System.out.println(currentValue );
```

Listing 2: Simplified Print Statement

When less complex source-code is provided (adding variables via the stack in lieu of the compounding expression), the JOP is able to deploy more efficient bytecode to take advantage of its enhanced stack operations and improve program throughput. In this case, fewer calls are made and therefore less bytecode is generated (note the number of *invokevirtual* and *invokespecial*). The concept of bytecode folding is discussed more in Section D.6. Similarly Schoeberl also introduced binary enhancements to the *new* command that defines and populates variables in a single loop (see the JOP handbook for further details [25]).

Most desktop machines provide an ecosystem to host or emulate a VM. The JVM interacts with middle-ware, the Operating System (OS) and kernel prior to executing code on the physical machine. A Java processor executes bytecode as machine instructions in hardware, all without the need for any added interpretation, Just-in-Time (JIT) processing or cross compilation. The JVM within the silicon chip is the machine and operates without a kernel, OS or middle-ware. Using the JOP, Java software can be written to execute application or serve data deterministically (with known physical constraints). This concept is demonstrated in Figure B4.

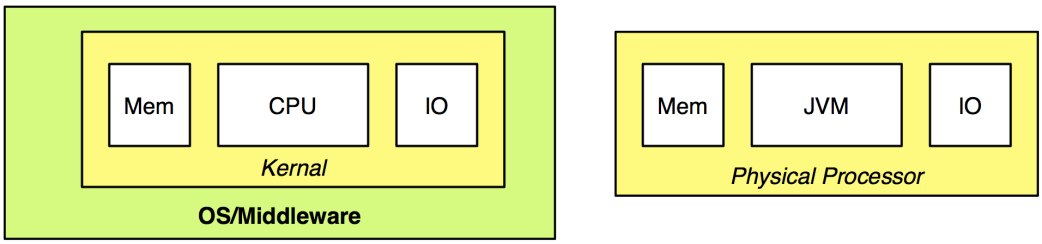

Figure B4: Virtual or Physical

This concept shows how a traditional JVM driven via a JIT compiler hosted by a PC can be transformed into a physical device, free of an OS, that becomes the machine! Additional functionality can also be incorporated physically to provide the same functionality associated with many middle-ware products.

B.3 Memory

The JVM provides a physical garbage collection that is capable of dynamically managing the memory pool of instantiated data types. Traditionally the memory management unit

on current Central Processing Units (CPUs) is a physical device that is software controlled via the kernel under the direction of the operating system monitor processes [9]. Using a JOP to physically embed the functionality of a JVM within the CPU architecture, removes the need for a kernel or OS. These components are redundant as are the delays associated with the monitoring activities. Unfortunately the functionality can vary because the memory footprint for Java changes according to the complexity required [46]. The capability of this functionality is classified using four categories. These are:

Java 2 Enterprise Edition (J2EE): There are 100's of classes associated with Enterprise Java Beans (EJB) and other enterprise classes. The default footprint for EJB 3.0 is over 100 MB of memory, without services running.

Java 2 Standard Edition (J2SE): The jar files for Java 2 Standard Edition (J2SE) amounts to over 15 MB of SOC memory, ROM or Hard Disk Drive (HDD) storage.

Java 2 Micro Edition (J2ME) + CDC: The footprint for CDC and kilo-Virtual Machine (KVM) still requires at least 2 MB of SOC memory (excluding the OS).

Java 2 Micro Edition (J2ME) + CLDC: This limited form of CDC using a KVM requires only 450 kB of SOC memory (excluding the providers OS). This is typically described as Connected, Limited Device Configuration (CLDC).

The only way to reduce the footprint for any particular implementation is to reduce data types, constrain the implementation of a given profile, pre-load classes within the JVM or to discard any redundant classes in lieu of existing functionality. Other complications occur when trying to invoke additional features such as a Real-Time Specification for Java (RTSJ) model. In embedded machines memory becomes immortal[48] and can eventually cause capacity constraints or even complete failure. Flexibility comes at a cost; either to the footprint and performance or the cost of analysis and functionality. The only real answer is to include hardware to implement concurrent garbage collection using instance counts and status bits as shown in the JOP hardware [47].

B.4 Garbage Collection

Studies have shown that the life of data (an Object) is relatively short [48]. Garbage collection is designed to dispose of redundant objects in order to free memory and promote reuse to reduce the volume of total amount of memory required on-board the system. By examining a *heap* stack with kbytes of memory, at least 90% of all newly created objects expire rapidly. It is feasible to easily reduce memory overheads by simply reusing the expired capacity within the allocation *pool* and an engineer can easily lower the overall memory budget of any system using a more efficient memory management process. The JOP employs a generational system to manage its memory *pool*. Other efficiencies are realised through bytecode folding and cache optimization (Section D.4).

[48] Embedded machines generally operate without operator influence making any data present in memory immortal. Most SOC products contain a combination of both static and flash memory, so even if the power is removed, some memory needs to be manually purged or re-flashed.

B.5 Real-Time Specification for Java (RTSJ)

At present developers have relied on Ada and C++ to support real-time applications for mission critical systems. We are generally forced to make choices (sacrifices) to achieve deadlines, and given the chance, benefit from reviewing alternatives. Effort to incorporate RTSJ into Java has evolved. This evolution is shown in Figure B5[49].

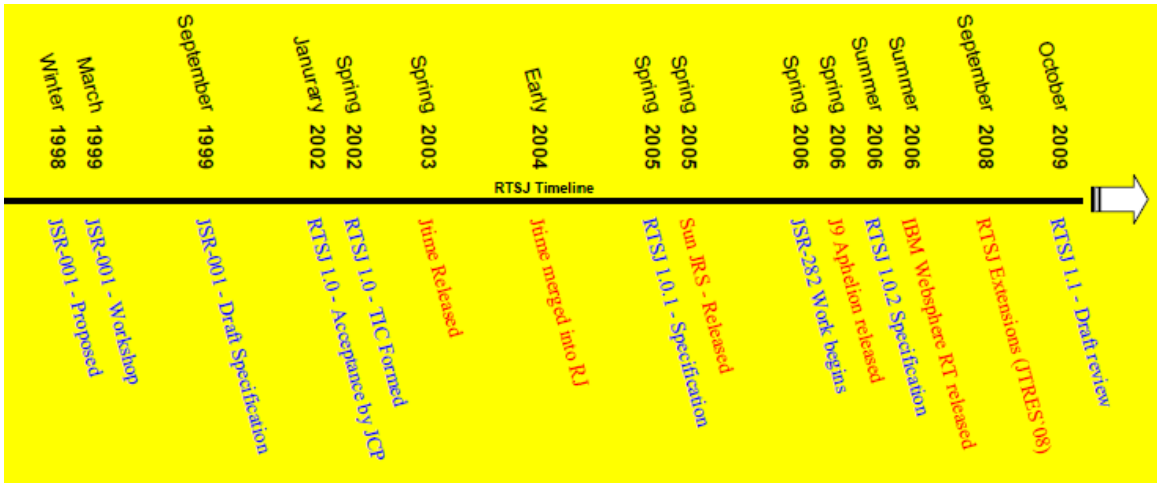

Figure B5: RTSJ Evolutionary Time-line (Updated)

The ultimate goal of its designers is to provide RTSJ software with service-like interoperability. The goal of this report is more modest. It begins with a literature survey on the origins and evolution of Java, prior to assessing the feasibility of using a dedicated (physical) processor to interface field-able systems in real-world environments by sampling all available research.

RTSJ is commonly used in applications that require hard timing constraints, such as robotics and critical systems. Recent work at Lund University discusses the use of RTSJ with real-time garbage collection employed to control the scheduling [47] of a *Flexpicker* pick-and-place industrial robot[50] using its native EtherCat control system. This is achieved using a standard platform based controller driven solely by Java code. The solution was demonstrated at Java Technologies for Real-time and Embedded Systems (JTRES) conference in 2007. The developers used real-time multi-threaded control loops for each axis of motion. These Proportional Integral Derivative (PID) controllers interfaced directly to the servo control mechanisms. Motion controllers generated trajectory information using velocity and torque drive controllers, all written in Java with a minor amount of Java Native Interface (JNI) code [49]. This demonstration successfully translated a video image of passing participants and repetitively captured their portraits physically on canvas in a timely manner.

The most relevant use of RTSJ in an airborne mission system was by Boeing in the *ScanEagle* Unmanned Air Vehicle (UAV) [27]. Other developments include work by Masson and Midonnet (at JTRES'08), who proposed extensions of the specification which enabled the

[49] See the RTSJ website at http://www.rtsj.org/
[50] This was a parallel kinematic robot (ABB IRB 340).

management of *aperiodic events* [50]. More recently, a draft review of the specification has been released under Java Specification Requests (JSR) 282 (release 1.1 at alpha 6.0).

THIS PAGE IS INTENTIONALLY BLANK

Appendix C Physical Virtual Machines

Following the conceptual aims of Pascal to execute p-code directly, three physical Java processors were initially proposed by Sun Microsystems to enable manufactures to build machines that would run bytecode natively. This concept continues to attract substantial debate and the reader is free to examine material from various domains before developing any conclusions. No discussion is afforded to commercial developments because they essentially promotional material or closed Internet Protocol (IP) solutions[51]. Several examples include:

- Timesys (initially funded by IBM);
- Jasmaica; and
- Ravenscar.

Given that this report is about the silicon instantiation of the interpreter, discussion remains focussed on how the Java Virtual Machine (JVM) was progressively abstracted. Hence the following description starts with a brief review of the kilo-Virtual Machine (KVM), moves on to the *picoJava*, the *microJava* and the *UltraJava* [20], prior to exploring the JEM, aJile, JavaCard and concluding with a list of other notable activities found in the literature.

C.1 The kilo Virtual Machine

Initially the term kilo-Virtual Machine (KVM) was used to highlight the memory budget and relationship between the JVM and traditional desktop platforms. This reference is with respect to only requiring kilo-bytes of memory to service a JVM as opposed to mega-bytes for most desktop applications. The term slowly faded, but is still referenced when discussing the core operation of *JavaCard* solutions. The intention was to create a compact, portable JVM that was specifically designed from scratch to operate small resource-constrained devices in as little as one hundred kilobytes of memory. More specifically, the KVM was designed to be [35]:

- small, with a static memory footprint of the Virtual Machine (VM) core in the range of 40 kilobytes to 80 kilobytes of memory,
- clean, well-commented, and highly portable,
- modular and customizable, and
- as *complete* and as *fast* as possible without sacrificing the other design goals.

It was originally targeted to operate on 16/32-bit Reduced Instruction Set Computer (RISC)/Complex Instruction Set Computer (CISC) microprocessors with constrained memory footprints, such as mobile phones, pagers, Personal Digital Assistants (PDAs), and Electronic Funds Transfer Point-Off Sale (EFTPOS) terminals.

[51] These designs have been targeted to specific Field Programmable Grid or Gate Array (FPGA) technologies with dedicated routing or customised Logic Cell (LC).

C.2 PicoJava

The term KVM faded with the introduction of the picoJava. The design was first announced by Sun Microsystems in 1997 [51]. It was introduced as a Verilog library to be implemented as an Application-Specific Integrated Circuit (ASIC) [52]. The picoJava was licensed by Fujistsu, IBM, LG and NEC, but was never commercially released, although it did attracted limited academic interest. For instance, Dey et al. recorded their experience with the simulated core using *model sim* [53]. They described their research in terms of using a System on Chip (SOC) attached as a co-processor design directly connected to the *PI-Bus* for hardware acceleration. Dey also reported the implementation consumed 440 thousand gates. The picoJava processor was eventually released to the public as an open-source project in 1999 [54]. Sun Microsystems also released reference manuals for both the *Micro-architecture* [55] and *Programming Reference* [56]. The basic design used a CISC-based processor core with over 300 instructions. This information enabled developers to implement enhancements and release the picoJavaII as prefabricated ASIC chips and more recently transition their libraries to FPGA chips. The picoJava microprocessor was composed of a number of core components, interconnected using internal data and control buses. Figure C1 shows the general configuration of the processor design. These include:

- Instruction Cache,
- Data Cache,
- Stack Cache,
- Integer Unit, and
- Micro-Read Only Memory (ROM).

All microprocessors require registers to monitor the physical state of operation within their environments. Examples include program counters, memory addressing registers and interrupt mechanisms. Each architecture uses a proprietary design and many rely on the programmer to consider these issues when creating efficient software (usually via the compiler). The picoJava is not based on a RISC architecture, but does make efficient use on single cycle instructions [52][52]. The hardware uses interrupts and traps to manage many of the more complex software tasks, such as creating arrays and invoking methods [57]. As a direct execution engine must be designed to cope with expensive resource intensive operations, especially when the hardware creates objects, manages arrays and conducts garbage collection [9].

The picoJava is a stack-based 32-bit microprocessing core with a variable length Instruction Set Architecture (ISA) of 300 instructions[53] implemented using a 6-stage pipeline. It was released with the option of including a physical or virtual Floating-Point Unit (FPU).

[52] In all microprocessors, many instructions relate to constants, local variables and stored values. In the picoJava these represent 15.2%, 41.5% (both predominantly stack based) and 24.2% respectively.

[53] A large number of multi-byte instructions are resolved at run-time and substituted by reserved single-byte alternatives that are only accessed by the core [52].

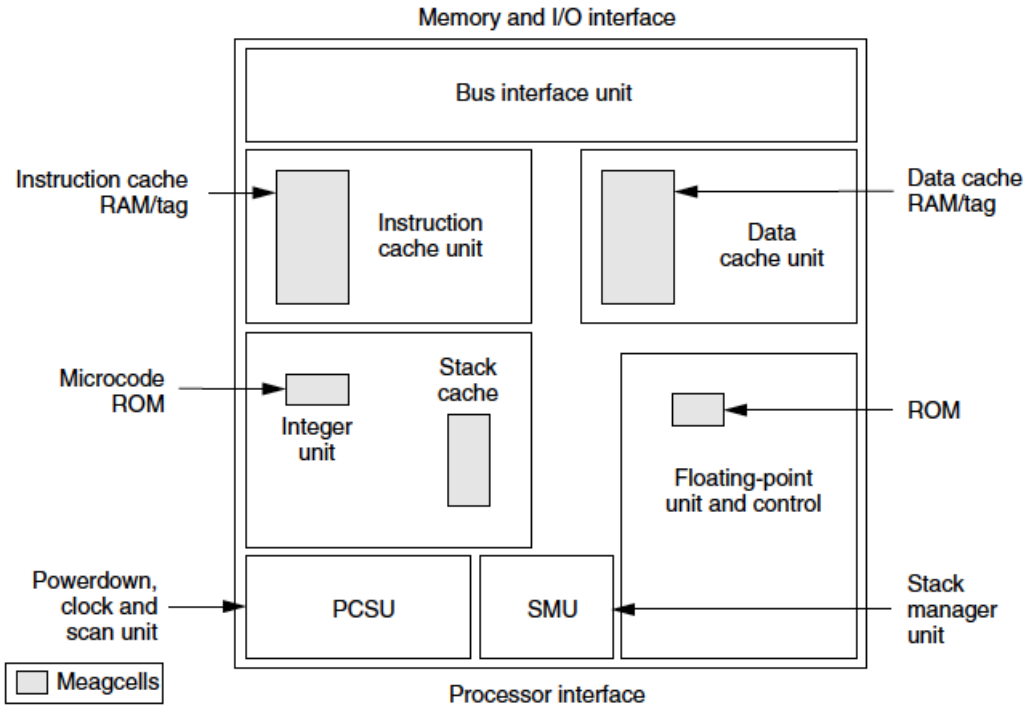

Figure C1: Block Diagram of the picoJava core [55]

The biggest impediment with this design is migrating the closed IP designs for the caches, integer unit, stacks, FPU and micro-ROM instructions to other environments[54].

The picoJava is an RTL hard core[55] that was designed for ASIC fabrication using Verilog code[56], similar to Digital Signal Processors (DSPs) and many router switching chips. Experience testifies that generating designs on different chips is a non-trivial process. Hence the user will generally need to generate and test the necessary masking image. A soft core is one that provides the IP of the design in a modular format that can be targeted with some flexibility to an given manufacturers technology. Common examples include Altera, Lattics and Xilinx FPGAs. A firm core on the other hand is one the that tries to balance the need for efficiency and optimization with that of flexible reuse [53]. Unlike the picoJava, modern designs provide gate-level libraries without the physical limitation or constraints that prevent the proliferation of many existing designs. This IP stifles the expansion of open source development due to cost, availability and licensing.

[54]There are a number of other considerations relating to the physical Register Transfer Level (RTL) footprint, implicit built in test and signal monitoring (Using primitive forms of Joint Test Action Group (JTAG) using Logical Built in Self-test (LBIST), Multiple Input Shift Register (MISR), Linear Feed Shift Register (LFSR) and Shift Register Sequence Generator (SRSG)) [58, 53].

[55]This is a synthesizable library with 46,376 Lines of Code (loc) in Verilog [53].

[56]The verification support for this design consisted of several libraries. The simulation library contained 22,454 loc, the runtime support 34,642 loc and the functional test another 11,348 loc [53].

C.3 PicoJavaII

Attempts to design a microprocessor that directly executes higher level instruction sets are not new. During the reign of Pascal, we had *pcode*. This was followed using the concept of *casting* Forth into silicon. This design also introduced the concept of using a *dribbler unit* to serve code without the typical *write-back* caching penalties. In 1985 Chuck Moore designed the *Novix NC4000* microprocessor to directly execute *Forth*. Harris technology quickly followed with the *RTX200* and soon after the *M17* Minimum Instruction Set Computer (MISC) processor. The *picoJava II* was released by Sun Microsystems as the microJava701 in 1997. Sun Microsystems subsequently licensed the technology to Rockwell Collins, who created the *JEM1* in 1997 which was quickly spun-off as the aJile *aJ-100*. Advantel followed the concept but eventually created an independent design called the *TinyJ*. Fujitsu released a the *J-StaterKit* based on the *MB86799* in 2000 complete with an Real-Time Operating System (RTOS) called *JTRON* [59, 60, 57].

The picoJava embodies robustness, portability and a high level of security, where memory is treated as a black-box to protect against malicious code. As its predecessor, it is a stack machine that manages a fully compliant variable length set of bytecode instructions [61]. It was released with the option of including a physical or virtual (software emulated) FPU. The design includes verilog components to build the *Stack Cache*, *Memory with Input/Output (I/O)* and *Instruction/Data Caches* [61]. This option provided developers with a low cost method of making the technology portable and enabled them to integrate this design into customisable products. Figure C2 reveals there were no conceptual changes to the original design or specification.

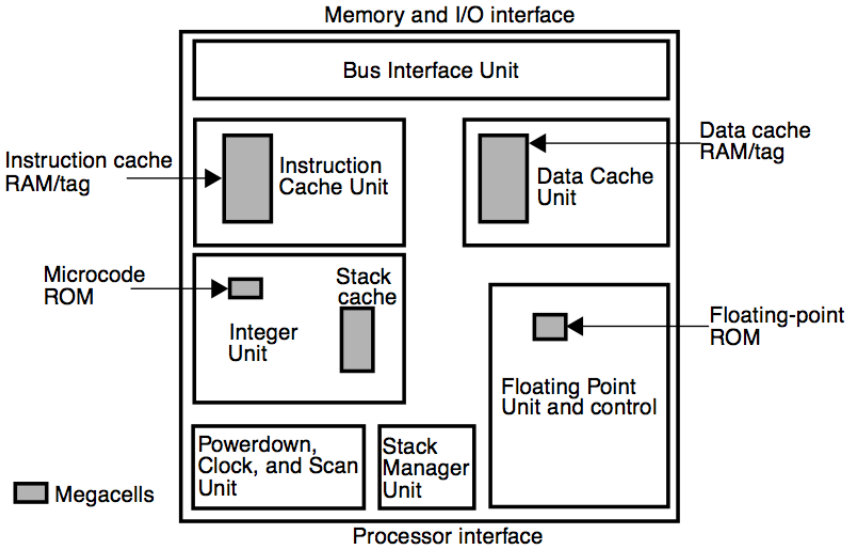

Figure C2: picoJavaII^{TM} Core Microarchitecture [62]

The PicoJavaII was originally released as a verilog code for ASIC designs in 1997 [52], which is no longer available, however a Fujitsu have released an FPGA as open source

project [63]. It also unveiled a real-time PicoJava processor clone called the *MB86799* in 2000 [59]. Many of these projects gave way to Just-in-Time (JIT) compilation embedded into Internet browsers.

C.4 JEM

Figure C3 shows the original approach to implementing the JVM by Rockwell Collins[57]. In 1998, Dr Jensen demonstrated he was able to create, document and transition Java into VHDL manually using the Altera MaxPlusII development environment. This approach enabled the transition from Access Control List language - Version 2 (ACL2) to provide higher-level language support through Java directly to the machine. He has continued this development of the resulting architecture into a commercially available version of the PicoJavaII.

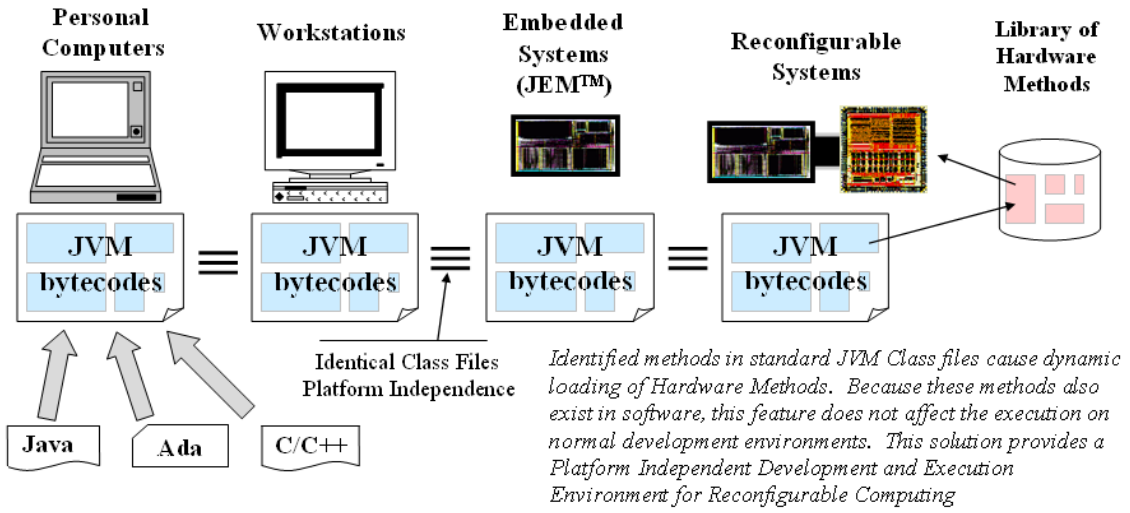

Figure C3: The Rockwell Collins approach to the JVM

The JEMCore enables the use of custom microcode. New instructions can significantly increase the throughput of frequently used algorithms or processes. For example multi-threaded instructions can be used to improve the processor's performance. The *yield* instruction results in a thread-to-thread switch of one microsecond. This operation would typically take several micro-seconds in existing RTOS written in a high level language.

C.5 The aJile JVM

The JEMCore is based on the proven JEM architecture from Rockwell Collins. This core was designed to directly execute Java bytecode in real-time. This core improves the speed

[57] See presentation by Dr David W. Jensen at http://hokiepokie.org/docs/seminar98.ps.gz

and efficiency by directly executing the instructions. This eliminates the need for the overheads normally associated with Java interpreters and JIT compilers. Primitives such as *wait, yield, notify, monitor, enter* and *exit* are implemented as extended bytecodes. This eliminates the need for an RTOS kernel. This core can be deployed as a native Java processor or as an independent coprocessor. This concept was used in 3G cellphones (sim cards) and Java Cards (almost all electronic bank cards). Figure C4 displays the JEM core embedded into the aJile processor.

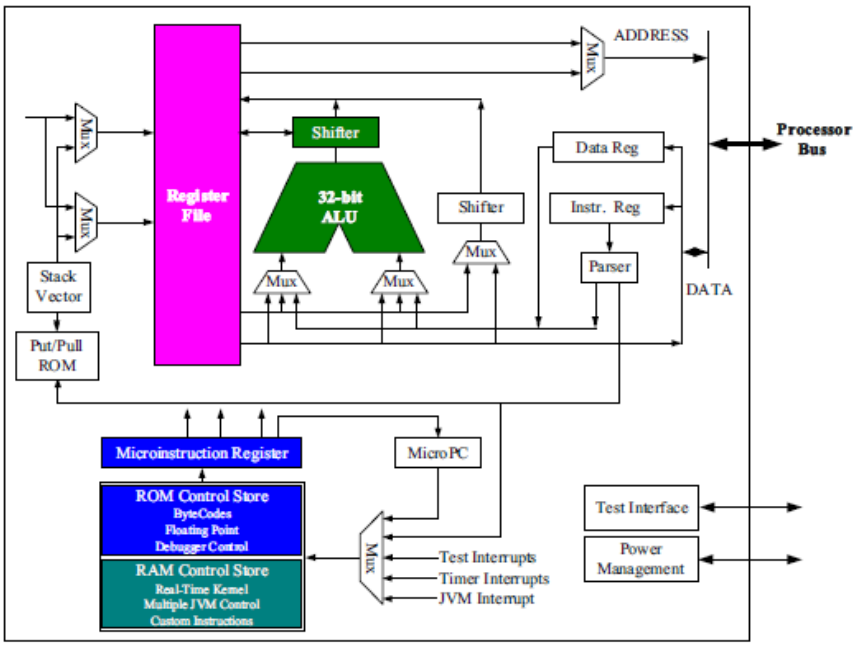

Figure C4: The JEM Core inside Rockwell Collins aJile Processor

The aJile processor has evolved from the humble aJ-80 through to the aJ-102. Using a multi-core design, the aJ-102 product provides a massive 300% performance boost over the aJ-100. This processor was positioned to radically enhance the platform intelligence of smart phone infrastructure into the 21^{st} century [64].

Danh Le Ngoc, who is the founder and Vice President of marketing for aJile Systems, believes the demand for networked SOC processors is being driven by the convergence of two powerful forces [64]. These include the expansion of mobile communications and networking. Examples include the ubiquity of 2.5G, 3G, Local Area Network (LAN), and 802.11 networks. Social networks and the growing desire for built-in intelligence are also forcing industry to invest billions in smart infrastructure.

C.6 JavaCard

A smart card is essentially a credit card sized SOC that stores and information. This silicon based virtual machine provides a portable, mobile computing concept for dedicated applications, such as security and banking. It is most commonly deployed as a customized

or dedicated product, such as an Asynchronous Transfer Mode (ATM) access card or electronic wallet. Early prototypes were implemented using 8-bit microprocessors based on a single-chips with embedded memory and private key encryptions. The JavaCard provides a static embedded system that is instant *ON* and has no need to dynamically load classes at run-time [65]. Unlike a computer a smartCard has NO support systems, such as a power supply, keyboard, hard drive or display. Its I/O relies solely on a physical serial interface. Originally JavaCards were provided as plastic cards that connected directly to a smart card reader[58] using a five pin mechanical connector. Sun Microsystems released a number of specifications relating to the JavaCard as it evolved, with version 3.0.1 adding wireless near-field communication [66] to the design.

In early 2010, an Oracle news release estimated there were over 5 billion access cards using embedded JavaCard processors[59]. Smart card technology today runs on 4 to 8 Kb of RAM and between 32-64 Kb of Electrically Erasable Programmable Read-Only Memory (EEPROM), however many still use slow 8-bit processors that only support a limited subset of JavaME.

The chip provides a JavaCard Run-time Environment (JCRE) which executes the native methods within the JVM and control access to any I/O in order to maintain system security of any embedded data[60]. Figure C5 displays the supported functionality of the Application Program Interface (API) and hard coded services[61]. There is also a primer[62] and other supporting documentation from the JavaCard website [63], Java Commerce website [64] and enthusiast publications (like Java World[65]).

C.7 The Java Silicon Machine (JSM)

As Java Cards became popular, the concept of more flexible Java processors evolved as the JSM [67]. More recent developments in physical the JSM promote embedded systems to support traditional applications. One example in the academic space is the Java Optimized Processor (JOP). A quick survey of notable research projects associated with the JSM has been extracted from the literature. These include:

picoJava: was the first attempt by Sun Microsystems to physically build a commercial Java processor [9].

aJ102 and aJ200: real-time low-powered network connected processor from aJile Systems [64].

[58] All smartCards can be inserted into a Card Acceptance Device (CAD) to exchange data.

[59] See http://www.itnews.com.au/News/229750,ellison-to-talk-javas-future.aspx?eid=1&edate=20100824&eaddr=

[60] This provides security for access to Personal Identification Number (PIN) numbers and access protocols.

[61] The JavaCard framework and associated developer kit is document by Oracle with the API Java Document at http://docs.oracle.com/javame/config/cldc/opt-pkgs/api/security/satsa-api/jsr177/javacard/framework/service/package-summary.html

[62] See http://www.javaworld.com/jw-12-1997/jw-12-javadev.html

[63] See http://www.javaworld.com/javaworld/jw-02-1998/jw-02-javacard.html

[64] See http://java.sun.com/products/commerce/

[65] See http://www.javaworld.com/javaworld/jw-02-1998/jw-02-javadev.html

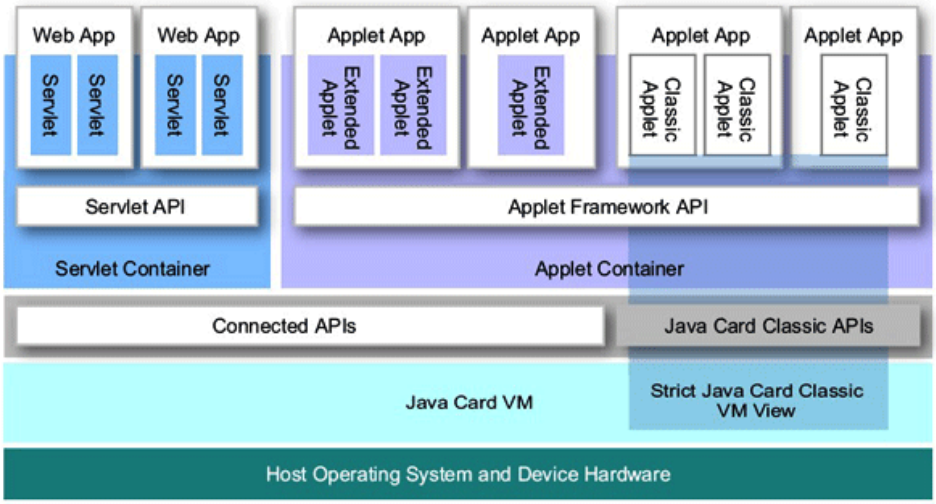

Figure C5: JavaCard [62]

Cjip: Reconfigurable System on Chip (RSoC) with embedded JVM from Imsys Technologies[66].

JA 108: Nazomi JSTAR and JSMART acceleration for Connected, Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP) products[67].

Komodo: is a multi-threaded Java micro-controller for research on real-time scheduling applications [68].

FemtoJava: is a research project to build an application specific Java processor [69].

ARM926EJ-S: A fully sythesizable Advanced RISC Machine (ARM) based[68] RISC processor able to run Java bytecode by *Jazelle*.

JOP: for FPGAs [70].

SHAP: A bytecode processor from the Dresden University of Technology in 2006 [71][69].

jHISC: provides hardware support for object-oriented functions [72].

ObjectCore: is a multi-core ARM based Java processor also designed by Vivaja Technologies originally using a Vertex V FPGA[70].

A microprocessor using a High-Level Instruction Set Computer (HISC) architecture, like jHISC[71], reflects the concepts previously developed by Phillips when implementing a Very

[66] Imsys is a Swedish fabless designer providing a rewritable-microcode chip with instruction sets for Java, Forth and C/C++. See the IM3910 - which is a High-Efficiency Microprocessor for Networked Equipment at http://www.imsystech.com/products/im3910.htm

[67] See http://www.nazomi.com

[68] See http://www.arm.com/products/processors/index.php

[69] See http://shap.inf.tu-dresden.de/

[70] See http://www.design-reuse.com/sip/pub-14516/, but you will need to register.

[71] jHISC is a 32-bit object-oriented processor.

Long Instruction Word (VLIW) design. This used a 128-bit operand descriptor to support Object Oriented Programming (OOP) with embedded machine readable data types. This concept is mainly used by Connected Device Configuration (CDC) products running Java 2 Micro Edition (J2ME).

THIS PAGE IS INTENTIONALLY BLANK

Appendix D Java Optimized Processor

Java can be cross-compiled, interpreted or directly executed given the appropriate resources. At present Just-in-Time (JIT) compilation has proven capable of implementing many serious enterprise level applications however, given the availability of System on Chip (SOC) suppliers, turn-key systems are increasingly being offered. Each requires specialist knowledge and experience in order to get products to market and reduce system modification which can be costly and take significant resources. Java is becoming ubiquitous and many applications can be fielded given a suitable platform capable of hosting a Java Virtual Machine (JVM). Size, mobility and flexibility are stating to drive distributed computing and autonomous control systems, which is influencing how products evolve. Customers are demanding systems that they can develop and modify using higher-level languages, such as Java, without the need to understand SOC design.

The Java Optimized Processor (JOP) has been manifested in many forms and on a variety of Field Programmable Grid or Gate Array (FPGA) based platforms. To move the design to a new target chip or development board, the hardware engineer simply change the memory and Input/Output (I/O) interfaces prior to recompiling the programming mask. This promotes re-use and reduces the need for the customer to modify any end-user applications. In many cases, it also enables a supplier to provide new hardware interfaces for a variety of systems within the same applications, departing from the need to rely on the existing low level design tools and technology. Several of the approaches requiring cross compilation, Ahead of Time (AOT), JIT, imaging and synthesis as shown in Figure D1.

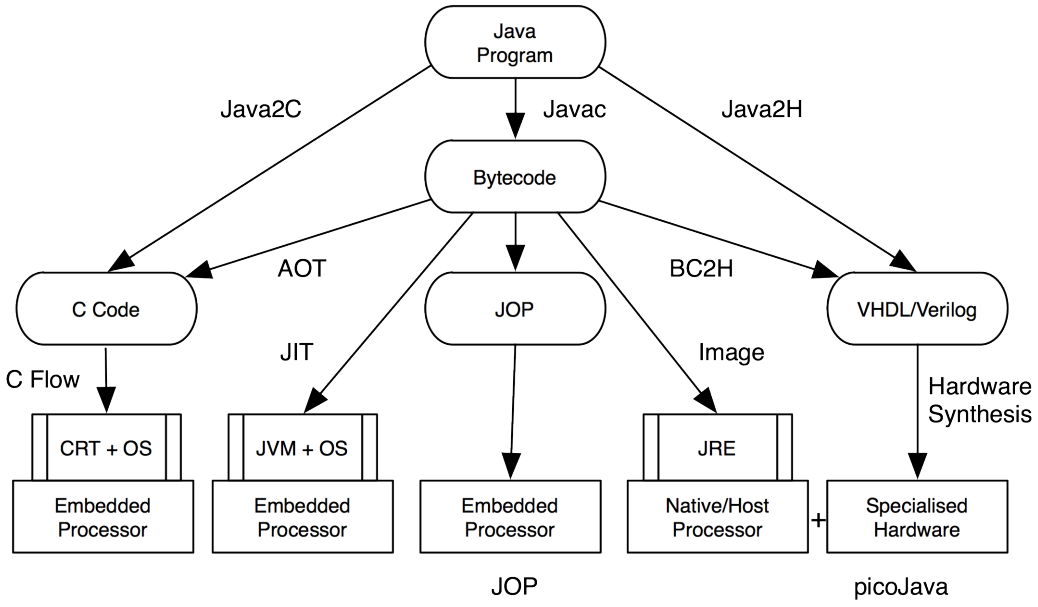

Figure D1: Approaches to Creating an Embedded Java Processor

Designs can be derived for existing hardware using cross-compilation or providing custom designs. These will be designed using either Application-Specific Integrated Circuit (ASIC) or FPGA methodologies. Examples of each approach include:

Co-processors:

- JSTAR from Nazomi in 2002 which also evolved as JSMART from the original Javacard development [73].
- Insilicon from Synopsys in 2001. This is based on an Advanced RISC Machine (ARM)9 using in hardware folding.
- Jazelle Direct Bytecode eXecution (DBX) from ARM in 2005[72].

ASIC Designs:

- Pica Java from Sun Microsystems in 1999 using a 6-stage pipeline with micro-programmed instructions.
- AJ-80 from Ajile using the JEM2 core on Connected Device Configuration (CDC) with Real-Time Specification for Java (RTSJ) compliance [74].
- CJIP
- FemtoJava was a research project that never passed benchmark testing [75].

FPGA Solutions:

- Moon from Vulcan which supports Java 2 Micro Edition (J2ME) Connected, Limited Device Configuration (CLDC).
- Lightfoot from DCT using a Xilinx Harvard core.
- Lavacore from Xilinx which is user configurable based on IP cores.
- Komodo from Augsburg University which provides four hardware threads [76].
- SHAP from Dresden University which includes garbage collection.
- JOP from Schoeberl providing a Complex Instruction Set Computer Complex Instruction Set Computer (CISC) approach with 4-stage pipeline and support for RTSJ [77].

The JOP is available as an off the shelf design and it can be employed using any number of existing development platforms. It can be customised during design however, when the product is fielded, the system will execute bytecode directly. The following sections provide more detail about the JOP design, including a description of the internal microcode, pipeline, cache, interrupt system and garbage collection.

D.1 Description

The major operational blocks of the JOP architecture is shown in Figure D2. The schematic looks similar to the picoJava, however significant effort has been expended to optimize several components within the JOP core. Stack-based machines traditionally take data directly from several sources (primarily the data stream and instruction queue). This process makes *opcode* throughput more efficient than the transactional approach used in CISC processors because the computational result is immediately accessible by subsequent instructions. As suggested, register based machines rely on a transactional process

[72] See http://mobile.arm.com/products/processors/technologies/jazelle.php?tab=Jazelle+Architecture

for every instruction. In CISC machines, most transactions use this process to takes information from memory, across a bus into a register prior to any calculation and then return data back to memory. It should be noted that because Java is written to support Object Oriented software, 65% of all bytecodes can be optimized within a physical JVM. Further optimization can also occur by manipulating access to the stack internally. For instance, you can physically bypass the need to negotiate a significant number of transactions via the Top of Stack (TOS) by accessing its contents directly using internally managed offsets.

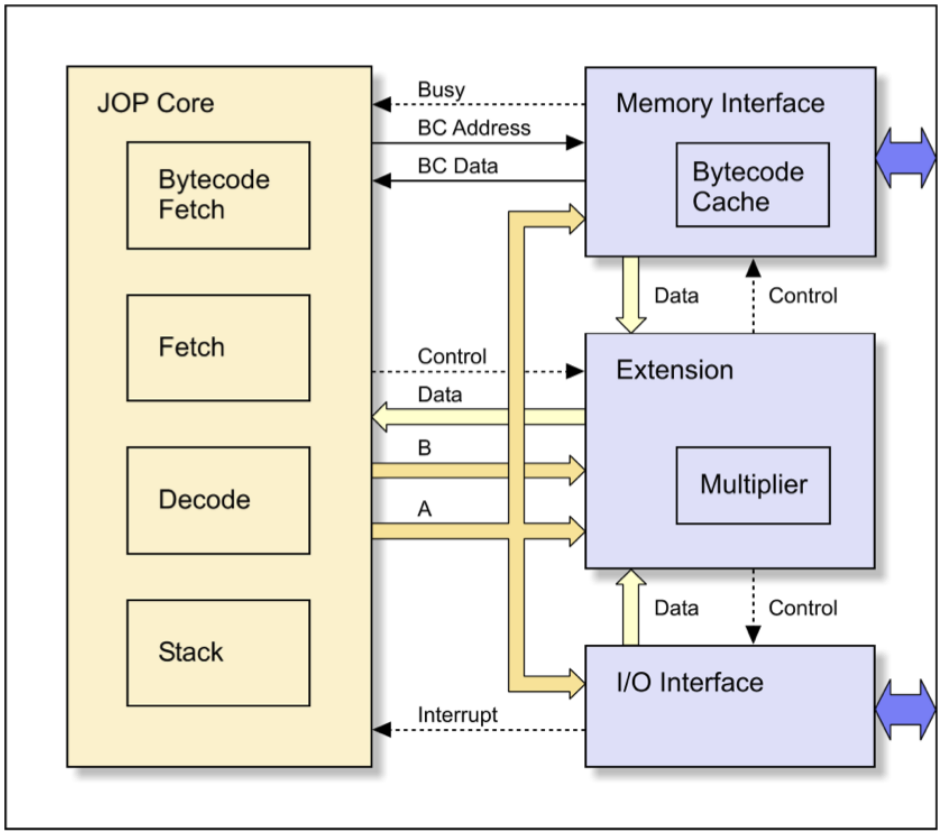

Figure D2: JOP Block Diagram [78]

D.2 Microcode

Like the original picoJava, the JOP has an extended Instruction Set Architecture (ISA). The standard set is exposed to the compiler and the extension's hidden to control the efficient operation of the machine. These are represented by the JVM supported *bytecode* instructions and the internal *microcode*. The latter is a native set of instructions used by the JOP to manage the extended stack and internal flow of the pipeline. As there are no bytecodes available to gain low-level access, these instructions transparently provide the functionality traditionally supported by an underlying Operating System (OS). It is also used to simplify the execution of some complex instructions [55] using *macro* style routines as *software traps* for simplicity (also described as *instruction folding* [79]). This enables the processor to internally substitute duplicitous instructions previously generated by the

interpreter. One obvious example is the **new** instruction. Other include:*invokestatic, invokevirtual* and *invokespecial* instructions.

A number of the Java bytecodes are extremely complex, however most of these are very rarely used. When they are used they can generally be re-assembled into more efficient forms. A series of 256 static methods have be specifically designed to optimized the stream for the JOP. This flexible implementation of bytecode is a supplementary step of the compilation process, however given sufficient resources can feasibly be included in the overall physical design [80].

D.3 Pipeline

As shown in Figure D3, JOP has a four-stage pipeline that executes bytecode instructions in a single cycle. The data path for both the *data* and *opcodes* are displayed in blue. The initial stage manages the bytecode stream similar to a monitor. It translates the instructions with machine readable meta-data into executable code together with any branching and storage logic. The next three blocks (second stage) *fetch*, *decode* and *execute* the micro-code generated. The third stage generates addresses to manage the stack *using the fill and spill technique* to enable it to *push and pop* data efficiently to the Arithmetic Logic Unit (ALU) [80].

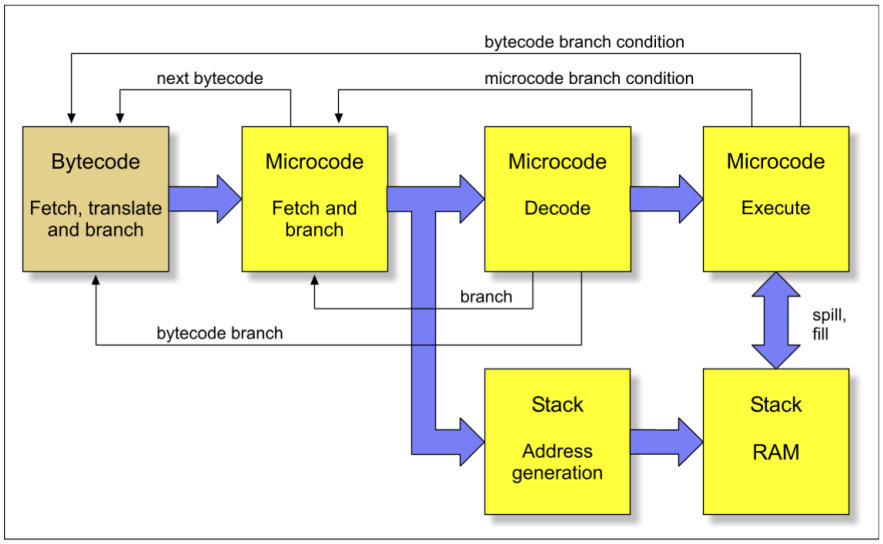

Figure D3: JOP Pipeline [80]

D.4 Cache

The JOP provides two time predictable caches [81]. One for the *instructions* and the other for *methods* [78, 82, 83]. The traditional memory bandwidth bottlenecks are minimized using on-chip Logic Cells (LCs) configured as block memory and is subject to microcode logic control to ensure it is accessible at predictable time intervals [84]. An internal call tree is used to manage the method cache. Complete methods are stored in the cache making

it easy to determine Worst Case Execution Time (WCET) and behavior. This concept provides a substantial performance improvement over traditional shared stack designs.

D.5 Interrupts

During translation the JOP cleverly avoids the need to handle interrupts within the pipeline. Special instructions have been provided internally to forecast an impending interrupt. These instructions are substituted to ensure interrupts always occur on byte-code boundaries. This enables you to determine pre-emptive delays for WCET timing calculations [85, 86, 87]. This form of mapping ensures that interrupts remain transparent to the pipeline core and minimise the design logic inherent in more complex systems. This solution provides a very acceptable solution for a very minor trade-off.

D.6 Bytecode Folding

JOP reduces code complexity through the use of Bytecode Folding. In essence this re-compilation process modifies (or re-orders) the program to improve access to the *stack* [88]. This is similar to the process described when introducing *bytecode* in Section B.2. When the *variable stack* is internal and independent of the *method stack* additional efficiencies can be realised. These concepts are still evolving and much will be written before a single standard emerges.

D.7 JOP Garbage Collection

One of Java's biggest criticisms is its efficiency related to creating and destroying objects [47]. To gain executing speed, objects are discarded and the resources then collected in a background process. This can cause sluggish behaviour during execution of many data intensive applications on machines with interpreted JVMs. To avoid this issue, Schoeberl intends on providing Real-Time Garbage Collection within the JOP [89, 90, 91]. He states that:

> A real-time garbage collector provides time predictable automatic memory management for tasks with bounded memory allocation rate with minimal temporal interference to tasks that use only static memory. [92].

Work is being conducted by his team to investigate supporting hardware to invoke write-barriers internal to the JOP aimed at optimizing the dynamic collection process to support real-time operation.

D.8 Multi-Threaded JVMs

Once upon a time, we commonly conceptualised a single Central Processing Unit (CPU) as being the machine. We also viewed the whole desktop as the CPU with peripherals.

Given the rise of multiple cores within a microprocessor chip, the desktop has become the machine and embedded systems are becoming computers. Soon multi-core computers will employ heterogeneous processor types (CPUs, Graphics Processing Units (GPUs), Digital Signal Processors (DSPs) and customized cores, like the JOP). In time, a multi-core JOP style SOC will perceivably become the computer. This would enable the JVM to transition from a virtual world to the physical world, enabling byte-code to run natively without the overheads introduced by the OS and delays caused by interrupts or multi-tasked processes (background daemons). Although multi-threaded application support on desktop machines is beginning to mature, they will continue to be constrained by the OS and middle-ware processes. The concept of having individual JVMs cooperatively interoperate using independent instantiations on a single Personal Computer (PC) have emerged [93], although interoperability can be stifled by the OS and associated software environment. At present the only way to support multi-threaded real-time applications is by adopting a vendor specific system that uses customised applications. Java SE-7.0 gained Portable Operating System Interface (POSIX) support. This enables Java programmers to enhance the scope of applications using a multi-core systems and illustrate the feasible use of multi-core Java processors, such as the BlueJEP and CMP.

D.9 BlueJEP

The BlueJEP Java processor is synthesised using a silicon core using a design process that is similar to the JOP as shown in Figure D4.

This uses the BlueSpec system Verilog process, which is a rule based, strongly-typed, hardware specification with a *Term Rewriting System* to describe the atomic state of computations. The design also incorporates BlueSpec Embedded Java Architecture with Memory Management (BluEJAMM) and has a redesigned six stage pipeline. The BlueJEP is significantly larger than JOP after synthesis, but it is claimed to exhibit slightly higher performance [49]. The microprocessor has the following properties [46]:

- Micro-Read Only Memory (ROM) program,
- Stack machine core,
- Real-time predictable system (in addition to being high-performance),
- Supports the full bytecode instruction set,
- Embedded micro-instruction set,
- Loaded and executes classes directly,
- Packaged with synthesis tools, and
- FPGA implementation.

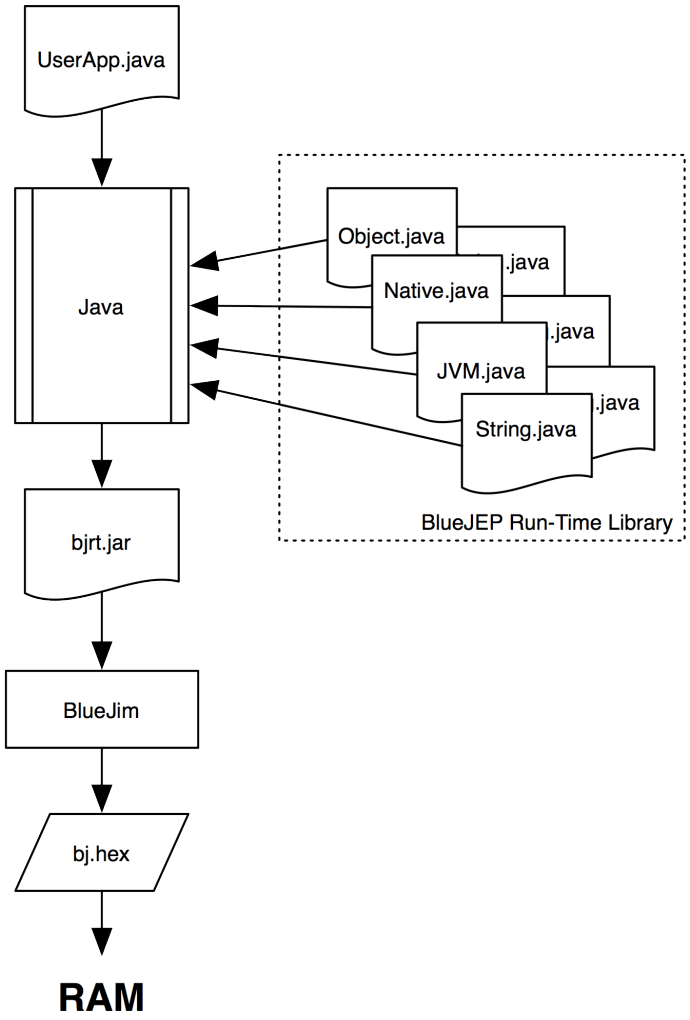

Figure D4: BlueJEP Flow

D.10 The CMP

Research into dual-threading Java processors emerged in 1992, with papers appearing periodically, such as the Simulataneous Multi-threaded (SMT) JavaChip [94]. A more recent example was examined by Pitter which employs Instruction Level Parallelism (ILP) across multiple JOP cores [95]. The Chip Multi-Processing (CMP) microprocessor block diagram is shown in Figure D5.

This design was successfully synthesized using Cyclone II (*EP2C70F896C6*) FPGA mounted on a *DE2-70* Development and Education Board using the Quartus II Integrated Synthesis Environment (ISE) as shown in Figure D6.

After confirming the success of the default design, an eight core array was synthesized. The results are shown in Figure D7. By reviewing the resources used in the flow summary, less than 5% of the chip's capacity was consumed. It is interesting to note that the:

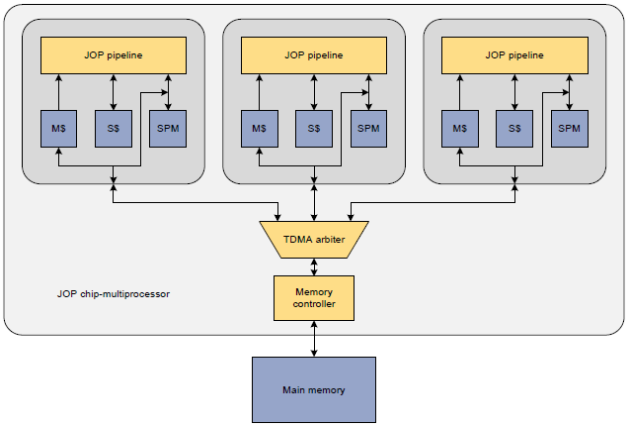

Figure D5: The Architecture of a Time predictable CMP [96]

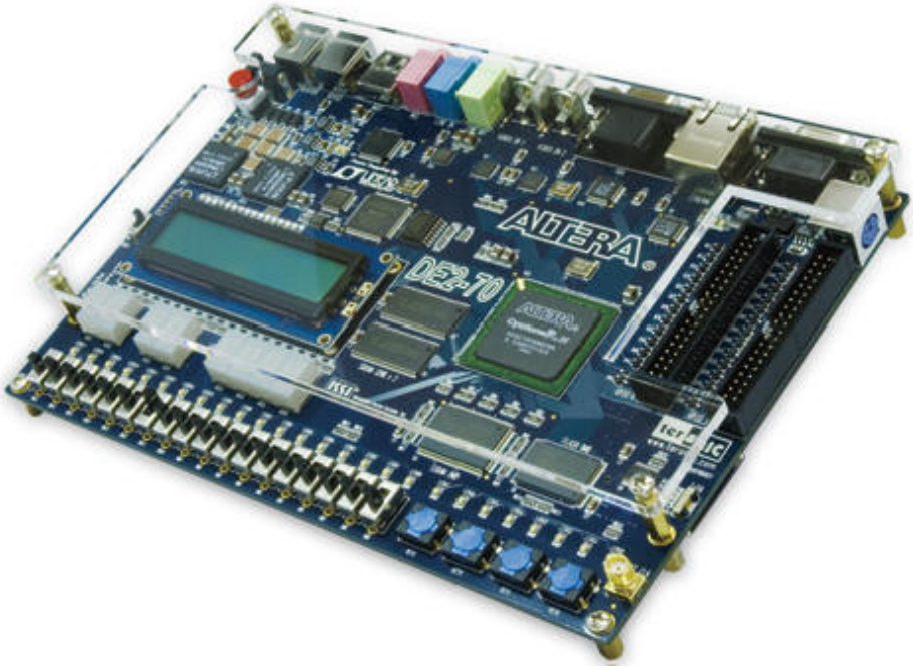

Figure D6: DE2-70 Development and Education Board

> "logic resource consumed for a stand-alone configurable design is in the range of 2000-3000 LC. That is 1/3 the size of a soft-core Reduced Instruction Set Computer (RISC) processor [97]".

A quick estimate indicates it is feasible to synthesize at least 160 cores in single chip design using the remaining capacity. Logic dictates that additional structure would be required, however the potential for creating a cluster of interconnected nodes does spur the potential for High Performance Computing (HPC) applications using a single SOC. For instance a 4 x 4 x 4 x 4 node cluster array (i.e. 256 interconnected cores) would deliver the performance of a powerful super computer on a single chip.

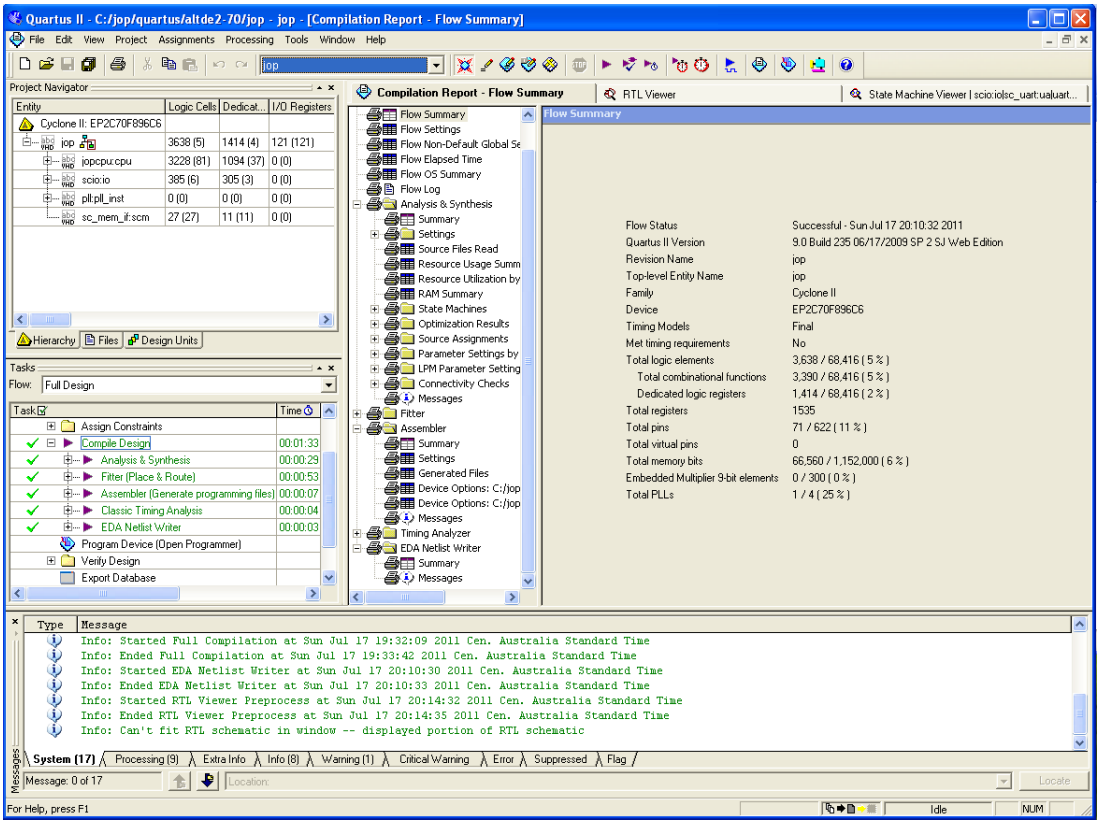

Figure D7: Results from Building an 8-core CMP project in Quartus II

As part of the research supporting this report, the ML-605 Evaluation board was also evaluated for a multi-core design. This boards was identified because of its potential of employing a large cluster of JOP as a SOC solution. As a guide, the board hosts a Virtex 6 *XC6VLX240T* FPGA with 241,152 LC building blocks. At this density, its not the number of Java Silicon Machines (JSMs) available, its more about the complexity of capabilities that can be interconnected in parallel.

Further documentation on the multi-core design is evolving at the University of Vienna with the latest investigating a time predictable design using a static Time Division Multiple Access (TDMA) schedule by Schoeberl [98]. The reader is invited to explore other contributions that should now start considering embedding the firmware protocols required to achieve interoperability. Industry concedes that Common Object Request Broker Architecture (CORBA) has a legacy following, however it has essentially been displaced by web-services and therefore recommend a compatible service-based interface be considered.

D.11 The Common Object Request Broker Architecture

The Object Management Group (OMG) originally released CORBA in 1991, however it was revised several times before an enduring version (3.0.2) was released in 2002[73].

[73] See the OMG History website at http://www.omg.org/gettingstarted/history_of_corba.htm

A revised standard (3.2) was released to accommodate near real-time functionality in 2011[74]. The Corba framework provides a skeleton interface for distributed interoperability of large homogeneous networks that supports the General Inter-ORB Protocol (GIOP), Internet Inter-ORB Protocol (IIOP) and Distributed Computing Environment (DCE) using Control Area Network Inter-ORB Protocol (CIOP) out of the box. It uses an Object Request Brocker (ORB) as a token that acts as a pass key to transfer a standard set of message formats or requests over different network using GIOP. This is an abstract protocol that specifies an implementation over Transmission Control Protocol / Internet Protocol (TCP/IP) using IIOP.

The real-time extension was provided to force computers with multiple microprocessors to support application in a predictable manner. This concept monitors execution priorities to map the underlying Real-Time Operating System (RTOS) task/threads being scheduled for all operations within the system. Figure D8 shows an ORB ended system that consists of network interfaces, operating systems, I/O subsystems, communication protocols, and CORBA compliant middle-ware components or services [99].

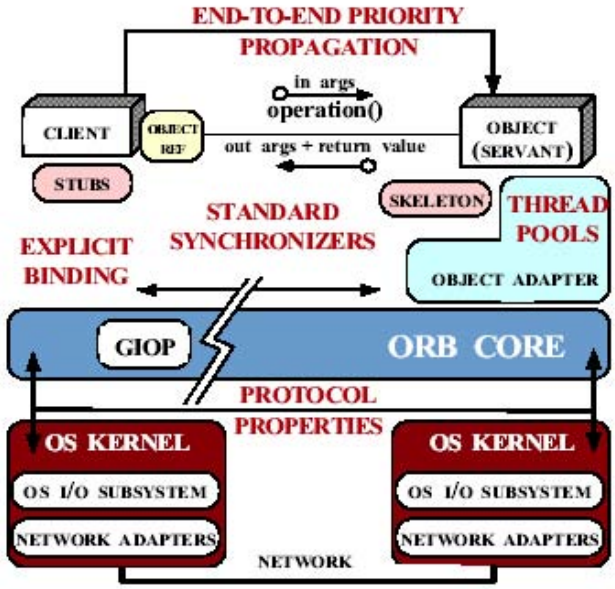

Figure D8: Possible Architecture in Real-Time Distributed Embedded Systems [26]

[74] See the OMG revised Specification website at http://www.omg.org/spec/CORBA/3.2/

DEFENCE SCIENCE AND TECHNOLOGY GROUP DOCUMENT CONTROL DATA			1. DLM/CAVEAT (OF DOCUMENT)
2. TITLE A History of Java's Progress to a Field Programmable Gate Array		3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED LIMITED RELEASE USE (U/L) NEXT TO DOCUMENT CLASSIFICATION) Document (U) Title (U) Abstract (U)	
4. AUTHOR(S) Jeffrey W. Tweedale		5. CORPORATE AUTHOR Defence Science and Technology Group PO Box 1500 Edinburgh, South Australia 5111, Australia	
6a. DST GROUP NUMBER DST-Group-TN-1723	6b. AR NUMBER AR-017-071	6c. TYPE OF REPORT Technical Note	7. DOCUMENT DATE January 2018
8. OBJECTIVE ID	9.TASK NUMBER	10.TASK SPONSOR DST Group	
11. MSTC Tactical Systems Integration		12. STC Human and Autonomous Decision Superiority	
13. DOWNGRADING/DELIMITING INSTRUCTIONS		14. RELEASE AUTHORITY Chief, Weapons and Combat Systems Division	
15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT *Approved for public release* OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111			
16. DELIBERATE ANNOUNCEMENT No limitations			
17. CITATION IN OTHER DOCUMENTS Yes			
18. RESEARCH LIBRARY THESAURUS Interfaces, Java Programming Language, Java Virtual Machine, Java Optimized Processor, Technological Innovation, System on Chip			
19. ABSTRACT This report documents a literature review of the methods associated with embedding a JVM into a Field Programmable Grid or Gate Array (FPGA). The review demonstrated that there was originally a commercial interest in providing silicon solutions for embedded JVMs, however with the exception of the JavaCard (which is embedded in all credit cards), few actually survived as mainstream products. To fabricate this form of design the developer requires a highly multi- disciplined team, that includes skills in programming, OS design, FPGA development and a signicant experience with very-low level hardware design. This report was written primarily to explore the existing domain by documenting the history, developments and possible future for this form of technology. A description of the basic concept together with more advanced applications have been described as example uses of this technology. Ultimately Defence Science and Technology Group (DST Group) could exploit the reuse and exibility of this approach to interface with legacy systems or exploit their interoperability for Defence.			