WHITE PAPER

ARM®

Jazelle for Execution Environments

New execution environment hardware support for compilation techniques

Chris Porthouse May 2005

The security, portability and improved time to market benefits arising from the use of execution environments (sometimes referred to as managed runtime environments) and “virtual machines” has led to their significant popularity across many embedded markets. Most notable has been the adoption of Java in applications such as wireless handsets, smart cards, set top box and digital TV, and even automotive applications.

While the prevalence of the Java language, development tools and large developer community has been pivotal in its widespread adoption in embedded, other languages and execution environments, such as Microsoft .NET MSIL/Compact Framework and Parrot (targeting Perl and Python) are also gaining in popularity.

Java adoption in wireless handsets has largely been driven by the growth in mobile gaming applications. As Java and other execution environments are installed on an increasing array of embedded devices, a significant proportion of applications deployed on these devices could be run on top of the execution environment. Increasingly, high performance is a key requirement.

Details of ARM’s first hardware for execution environments, the Java bytecode acceleration technology – Jazelle DBX (for Direct Bytecode eXecution), can be read in a separate white paper [1]. Jazelle DBX will continue to play an important part in ARM’s Jazelle for execution environments roadmap.

This ARM white paper explores the key technical issues involved in enabling performance and efficiency in execution environments. A new extension to the Jazelle family and the ARM Instruction Set Architecture (ISA) is introduced, and its technical features are outlined.

Execution Environment Efficiency

A program written in a language that is designed to run on a virtual machine (VM), is usually compiled to a pseudo-assembler bytecode language. This bytecode is then downloaded and executed on the target device within a virtual execution environment, such as a Java VM.

There are a number of ways in which the bytecode can be executed. The type of execution environment chosen will determine the performance that can be achieved and the memory footprint requirements.

Direct interpretation of bytecode in software is used widely in embedded Java applications today. This approach is effectively a software ‘switch’ statement which maps individual Java bytecodes to machine instruction sequences. This is efficient in terms of memory footprint requirements, but the interpretation process severely limits performance. ARM’s Jazelle DBX technology implements direct execution of Java bytecode in hardware, significantly boosting performance, as the bytecode effectively becomes the native instruction set. A key benefit of interpretation techniques is that execution of the application is immediate – there is no start-up delay.

Run-time compilation, often referred to as Just In Time compilation (JIT) or Dynamic Adaptive Compilation (DAC), is an alternative to pure interpretation. A JIT compiler first profiles the code to identify the most frequently executed functions. Once identified, this code is then compiled to the processor’s native instruction set and this code is run. Whilst executing compiled code can, in theory, give extremely high performance, compiling the majority of the program to native code can place significant demands on available ROM and RAM. The compilation process will typically result in execution code that is 2-8 times the original bytecode size. In addition, the original bytecode must be retained in case available memory is exceeded, requiring previously compiled code to be discarded and if it is required again, re-compiled from the original bytecode.

There are other issues with JIT/DAC compilation. Because the compiler runs on the execution machine in *user time*, it is constrained in terms of compile time: if the time taken to compile code is too long, then the user will perceive a significant delay in the startup of a program or in responsiveness while an application is running.

These user experience and memory consumption issues lead to compromises in the potential top-speed performance of a JIT/DAC solution, as advanced optimizations are simply not realistic, and the compiler must focus on generating compact code as opposed to speed optimized code.

Compilation of bytecode can also be done ahead of time (AOT). For example, code can be pre-compiled to native machine instructions. This can be done either before downloading onto a device over the air (pre-compile an application and store on a server for download), or it could be compiled and then installed on the device during manufacture (often referred to as “ROMisation”). Alternatively the compilation can be performed while downloading an application for the first time.

While AOT compilation can boost execution performance, if the majority of the application is pre-compiled, the demands on memory – ROM and RAM, can be significant. Also, AOT compilation alone can generate less optimal code than compilation at run-time; for example resolving methods in Java is often far easier and faster at run-time. Basically, the more that is known about the application, the better it can be optimized. In most cases, more is known at run-time than ahead of time.

Balancing Performance with Resource Requirement

For the very best and most efficient performance of execution environments, a combination of fast interpretation, run-time compilation and selective ahead of time compilation should be used. The level of compilation possible will be dependent on the resources available on the device – the CPU speed, available memory ROM/RAM and cache.

AOT and JIT compilation techniques can offer extremely high performance with execution environments. However, the memory overheads in generating highly optimized code are severe.

The memory available on devices such as digital TV and high end smart phones is increasing rapidly as memory costs fall. However, the usage of the available memory is also increasing rapidly: multiple application support and complex multi-media user applications are putting strains on even the highest specification embedded devices.

A typical Java execution environment must compete with imaging, video, audio and other applications on a device, with typically only around 10% of total system memory available to it. Clearly, efficient use of available memory by any execution environment is extremely important.

A New Architecture Extension

To enable improved compilation performance while managing the demands on memory resource, a new approach is required. ARM's Thumb-2® ISA provides the basis for balancing code density and performance through a blended instruction set combining both 16-bit and 32-bit instructions, and is the foundation for the new ARM architecture extension for efficient execution environment support.

ARM Thumb-2 core technology, extends the ARM architecture to add enhancements to the Thumb ISA that simultaneously benefit code density and performance. The resulting ISA consists of the existing 16-bit Thumb instructions augmented by new 16- and 32-bit instructions to improve program flow and performance.

Thumb-2 can now access all of the instructions it needs to enable both high performance and exceptional code density. In addition, using Thumb-2 core technology considerably simplifies the development process, especially when the trade-off between performance, code density and power is not straightforward. The main reason for this is that code 'blending' – changing the mix between ARM and Thumb instruction usage, is no longer necessary.

Ideally, an execution environment solution will provide good support for AOT and JIT compilation in a memory efficient way and without significantly increasing hardware gate count.

ARM's solution is a new addition to its Jazelle family for execution environments: Jazelle RCT (Jazelle **R**untime **C**ompilation **T**arget). Jazelle RCT is an extension to the ARM architecture featuring a new ISA that extends Thumb-2: Thumb-2EE, and a new processor state: Thumb-EE.

Jazelle RCT benefits from the code density and performance features that are inherent within Thumb-2 core technology and provides an ideal target for "bytecode" languages like Java,.NET MSIL (Microsoft Intermediate Language), Python and Perl with a compilation target which is within 10% of the original bytecode size.

A runtime compiler (AOT or JIT) using Jazelle RCT can match (and sometimes better) the performance from a Thumb-2 AOT solution with almost no increase in compiled code size from the original bytecode. The additional hardware requirements to implement the solution are negligible in terms of gate count (less than 8K gates) and power consumption.

Compilation Support

Today, virtual machine developers focus much effort on generating compact compiled code, rather than high performance code. The primary aim with the design of Jazelle RCT has been to achieve efficiency in JIT and AOT compilation. Attaining reduced target instruction size enables the virtual machine developer to focus more on performance, rather than code size.

Jazelle RCT provides an excellent target for run-time compilation approaches including JIT and AOT, making these techniques more appealing because they can be realistically implemented in embedded systems. The Jazelle RCT implementation is a unique approach which leverages more of the architecture's inherent capability, and can be thought of as "Thumb for VMs".

Jazelle RCT Technical Overview

The new Jazelle RCT instruction set architecture is based on the extended Thumb-2 ISA. It adds 12 new instructions to Thumb-2 and modifies the behaviour of some existing Thumb-2 instructions and is accessible through entering a new state 'Thumb-EE'. Because Thumb-2EE is based on a modification of the existing Thumb-2 instruction set, silicon size and complexity is not significantly impacted.

The new processor state, 'Thumb-EE', is entered and exited with the new ENTERX and LEAVEX instructions.

The 16-bit load/store multiple (LDMIA and STMIA) instructions are no longer available, which was necessary to create the space for new Jazelle RCT instructions, however all other existing Thumb/Thumb-2 instructions continue to be available (note, some loads/stores also have modified behavior to improve performance on execution environments). It is therefore expected that an execution environment will not need to switch between Thumb-2 and Thumb-EE states frequently, but stay in Thumb-EE state for most of its execution.

A combination of techniques helps to achieve the code size reduction which results in a small compiled code footprint. Key to this is extensive use of 16-bit instructions. Almost every instruction generated by a run-time compiler for Thumb-EE can be a 16-bit instruction. This efficiency is enabled by extending the use of registers so that more 16-bit instructions can be used.

A general purpose 16-bit Thumb or Thumb-2 instruction can only access R0 to R7. The Thumb-EE state provides new 16-bit instructions that allow limited access to R8 and above for specific purposes, such as the Java stack pointer and constant pool. Run-time compilation is then free to use R0 to R7 for caching stack items and local variables. Only 16-bit instructions are required to do this – thus reducing code size.

Additionally, for frequently used code sequences in a JIT/AOT compilation, single 16-bit versions of instructions are available.

There are two enhancements included in Jazelle RCT that lead to direct performance improvement:

- Implicit Null Pointer tests: Null checking of objects and arrays in Thumb-EE.
- Fast Array Handling. A Thumb-EE instruction is provided to help with array index checking.

While Jazelle-RCT offers features that aid performance directly, the primary performance improvements will be due to enabling the compiler to focus on generating fast code, leaving Jazelle RCT to take care of code density.

Jazelle RCT Adoption Roadmap

Jazelle RCT will be introduced across a range of Thumb-2 compliant ARM cores, initially supporting Java on higher-performance devices.

Jazelle RCT will be an integral part of the ARM Cortex-A Series, ARM's applications processors for complex OS and user applications. Jazelle RCT will be optional on the ARM Cortex-R Series - embedded processors for real-time systems. However, since the ARM Cortex-M Series targets deeply embedded processors optimized for cost sensitive applications, it is not appropriate to make Jazelle RCT available on this processor series.

In parallel, ARM is working with lead software partners to ensure smooth adoption of Jazelle RCT.

Jazelle DBX or Jazelle RCT

With both Jazelle DBX and Jazelle RCT available, ARM provides a choice of support for execution environments which is appropriate to the needs of the application and can be matched to the capabilities of the underlying hardware platform.

Table 1 illustrates the applicability of Jazelle DBX and Jazelle RCT. Choice is dependent on the available system resources and the desired performance level.

Bytecode Execution	**+ Advantages** **- Disadvantages**	**ARM Solutions**
Direct Interpretation	+ Efficient memory usage + No start-up delays - Can be slow (performance)	**Jazelle DBX** + Direct hardware execution boosts performance - Java only support
Run-time Compilation JIT Compilation (or DAC) Profile code and compile most frequently executed sequences	+ Good performance - Increased memory requirements - Start-up delays	**Jazelle DBX** + Direct hardware execution boosts performance for interpreted code + Fast start-up + Memory efficiency – less compiled code **Jazelle RCT** + Efficient compilation addresses code bloat for compiled code + Enables improved performance + Supports several VM technologies
Ahead of Time Compilation Pre-compile entire application ahead of run-time	+ Good performance - High memory requirements - Difficult to compile efficiently ahead of run-time	**Jazelle RCT** + Efficient compilation addresses code bloat + Enables improved performance + Supports several VM technologies

Table 1. Code Execution Techniques: Advantages and Disadvantages

Both Jazelle DBX and Jazelle RCT may be implemented on future mid-range frequency ARMv7 processors to provide maximum flexibility in matching performance with the available on-chip resources. Thus, systems designers may choose

- Jazelle DBX for interpreter solutions only on a resource-constrained device,
- Jazelle RCT to support JIT or AOT compilation.

- Jazelle RCT and Jazelle DBX for JIT and selective AOT
- Jazelle RCT and Jazelle DBX , for selective AOT and an interpreter VM

Mid-range processors do not yet offer high ‘GHz’ performance, and so when running complex Java applications, both start-up compilation time and smoothness in performance are still issues that can be addressed with Jazelle DBX. With very high-performance, high-end processors, start-up delay (identified above as a problem with JIT/DAC compilation) becomes less of an issue, hence there is less need to offer Jazelle DBX with this class of processor.

Code Density and Performance Benchmarks

The main performance improvement from Jazelle RCT arises from the potential to focus on compilation for best performance, while the memory footprint of the compiled code stays within 10% of the original bytecode.

Jazelle RCT supports the capability to compile all installed classes and applications ahead of time, so no run-time compilation or profiling is strictly necessary. Profiling code inevitably consumes some cycles even if performed at a basic level by counting the Java methods which are regularly executed.

A common technique used to achieve better performance is inlining of methods. The additional code bloat from compiling to ARM or Thumb-2 is likely to make excessive inlining an expensive option on any platform. However, because of Thumb-EE’s small code size, selective or even extensive inlining may be possible in order to achieve very high performance.

In summary, Jazelle RCT enables the developer to choose between compiling for code density or performance. Compiling for code density gives extremely small memory footprint, while keeping performance levels as good as JIT compilation for Thumb-2 instructions. Compiling for performance will yield code size which is equivalent to T-2 compiled code, but performance can be far higher.

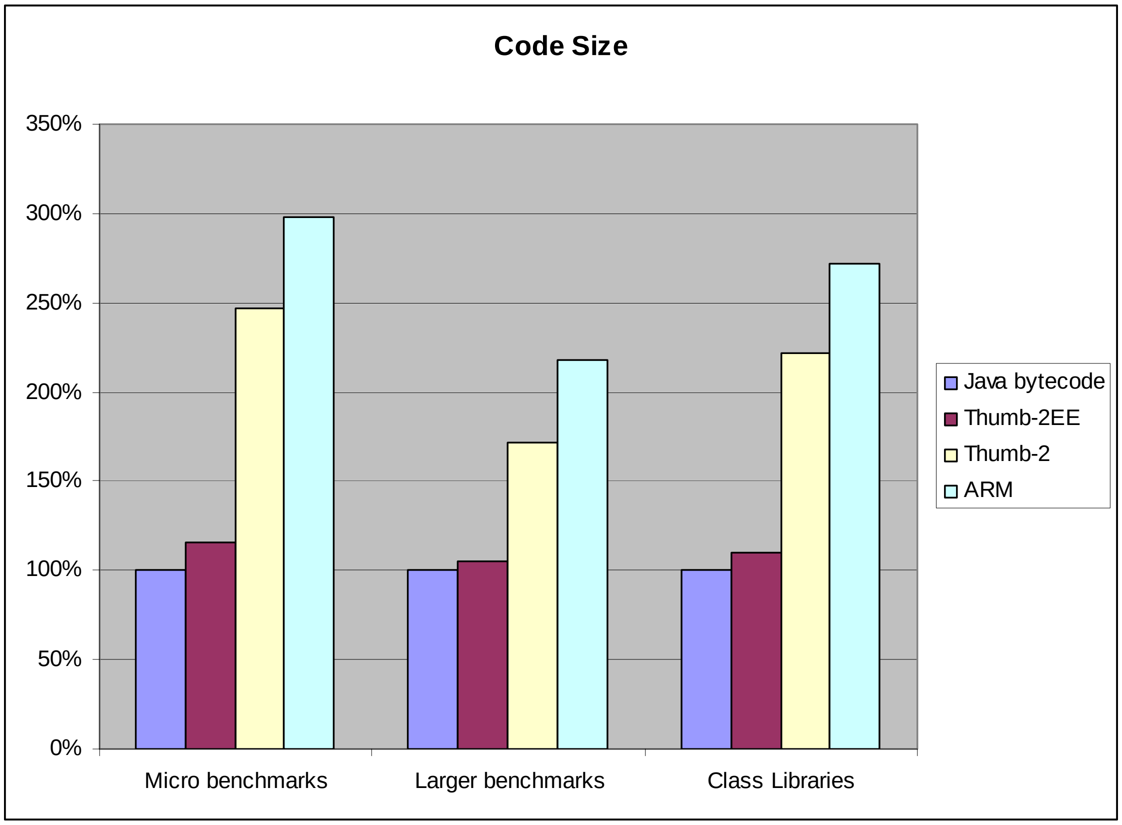

Figure 1. Preliminary Benchmark Results for Code Density

Figure 1 (Code Density) illustrates the results of a compilation of Java bytecode using an ARM AOT compiler. The compiler output includes ARM code, Thumb-2 code and Thumb-2EE (Jazelle RCT) code, and the results are shown normalised against the original Java bytecode. Note this compiler is highly optimized for code density rather than performance, so code bloat for all ISAs is at the low end of the possible range.

Compared with the original bytecode, the compiled Jazelle RCT code size varies from 1.07x (best case) to 1.44x (worst case). By comparison, the Thumb-2 code size varies from 1.89x to 3.08x for the same code examples.

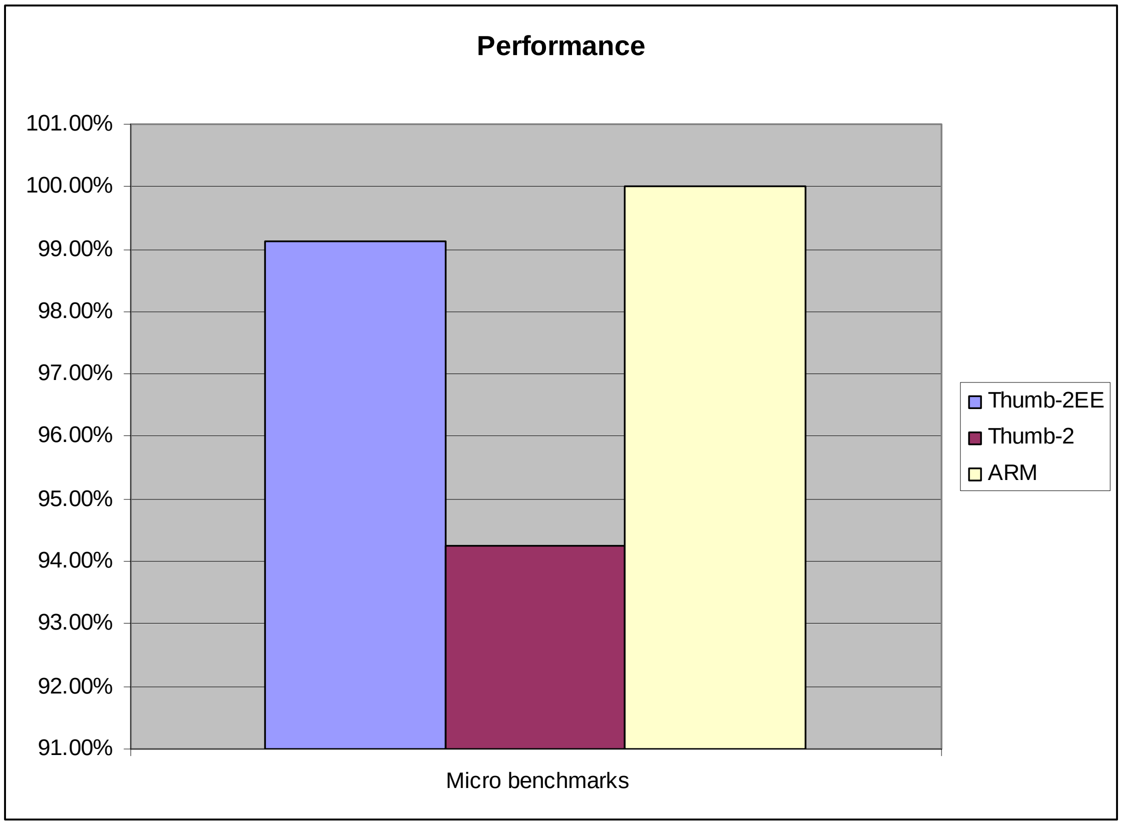

Figure 2. Preliminary Benchmark Results for Performance

Figure 2 (Performance) benchmark data is based on an ARM Jazelle RCT core simulation model run with a number of micro-benchmarks. Simulation results confirm that Thumb-2 requires 5-10% more cycles than the Jazelle RCT performance.

So, benchmark results confirm a code bloat figure which is within 10%, on average, of the original bytecode, yet performance is as good if not better than compiling for Thumb-2.

Thumb-2EE Instruction Set Features

The new features and instructions that comprise the Thumb-2EE instruction set extension are outlined in the following sections.

Null pointer checks

All loads and stores in Thumb-EE state check that the base register used for the address calculation is non-zero. If it is, the memory access does not happen, and execution continues from a null pointer exception handler, at address HandlerBase-4. HandlerBase is a coprocessor register configured by each individual VM, and is context switched by the OS.

Array bounds check: CHKA Rn,Rm

CHKA is a new 16-bit instruction, taking two registers in the range R0-R15. If Rm >= Rn (unsigned comparison), then execution will continue from an array index exception handler, at HandlerBase-8. If Rn holds an array size, known to be positive, and Rm an array index, the exception is taken if Rm < 0 or Rm >= Rn.

Handlers: HB{L} #handler

A new 16-bit instruction, which performs a branch, with optional link, to one of 256 ‘handler routines’, specified by the value of #handler. Execution continues from an address given by

HandlerBase + 32*handler

with a return address optionally stored in R14

HB{L} is branch predicted.

A handler routine will contain a commonly used sequence, often corresponding to a complex bytecode; for example, Java bytecodes such as idiv, fadd, lmul, athrow, and many more. Other uses include implementing a call to a profiling routine, or calling a thread switch routine for a VM that implements co-operative threading.

Handlers: HB{L}P #param,#handler

This instruction is a variant on HB{L}, and allows a small integer parameter be passed to a handler. There are some restrictions in usage – you can only call the first 32 handlers with HB{L}P. HBLP allows a parameter in the range 0 to 31, while HBP allows a parameter in the range 0 to 7.

The parameter is copied into R8 for the handler routine to use, and so does not corrupt the general-purpose low registers R0 to R7 available to a JIT/DAC**.**

This branch instruction variant can call routines that require a parameter that is known at compile time – e.g. the bytecode newarray <type>. The instruction can also be used for bytecodes that require an index into the constant pool e.g. new, all invoke bytecodes.

LDR Rd,[R9,#offset]

STR Rd,[R9,#offset]

These instructions load or store any low register to an address calculated by adding an offset to R9, where the offset is a word aligned value from 0 to 252. Typically, R9 would point to an area of memory in the stack frame used to store the current method’s local variables and stack spill.

LDR Rd,[R10,#offset]

This load instruction can load any low register from an address calculated by adding an offset to R10. The offset is a word aligned value from 0 to 124. Typically, R10 would point to the constant pool of the class of the current method

LDR Rd,[Rn,Rm,LSL#2]

STR Rd,[Rn,Rm,LSL#2]

These instructions replace the 16-bit

LDR Rd,[Rn,Rm] and STR Rd,[Rn,Rm]

This allows for Rm to be used as an offset into an array of 32-bit elements without having to use a 32-bit instruction or having to explicitly multiply it by 4 to get the correct memory offset.

LDR{S}H Rd,[Rn,Rm,LSL#1]

STR{S}H Rd,[Rn,Rm,LSL#1]

These instructions replace the 16-bit

LDR{S}H Rd,[Rn,Rm] and
STR{S}H Rd,[Rn,Rm] instructions.

This enables Rm to be used as an offset into an array of 16-bit elements without having to use a 32-bit instruction or having to explicitly multiply it by 2 to get the correct memory offset.

Sample Code

The code fragments below compare source code with Java bytecode and its compiled output in both Thumb-2 (Figure 3) and Jazelle RCT (Figure 4) code. The use of new Jazelle RCT instructions can be seen, as well as the comparative storage requirements for each instruction.

			Code Size	
Java Source	Java bytecode	Compiled Thumb-2	Java	T2
`X = new int[50];`	`bipush 50`	`MOV    R0, #50`	2	2
	`newarray int`	`MOV.W  R8,#T_INT`	2	4
		`BL.W   DoNewArray`		4
	`astore 5`	`STR.W  R0,[R9,#20]`	2	4
`X[index] = data`	`aload 5`		2	
	`iload 4`	`LDR.W  R1,[R9,#16]`	2	4
	`iload 6`	`LDR.W  R2,[R9,#24]`	2	4
	`Iastore`	`CMP    R0,#0`	1	2
		`BEQ.W  NullPtrHandler`		4
		`CMP    R1,#50`		2
		`BHS.W  ArrayIndexHandler`		4
		`STR.W  R2,[R0,R1,LSL#2]`		4
		Total	13	38

Figure 3. Sample Thumb-2 Code (*.w suffix indicates a 4-byte Thumb-2 instruction)*

			Code Size	
Java Source	Java bytecode	Compiled Thumb-2EE	Java	T2
`X = new int[50];`	`bipush 50`	`MOV    R0, #50`	2	2
	`newarray int`	`HBLP.X #T_INT, #NewArray`	2	2
	`astore 5`	`STR.X  R0,[R9,#20]`	2	2
`X[index] = data`	`aload 5`		2	
	`iload 4`	`LDR.X  R1,[R9,#16]`	2	2
	`iload 6`	`LDR.X  R2,[R9,#24]`	2	2
	`iastore`		1	
		`MOV    R7,#50`		2
		`CHKA.X R7,R1`		2
		`STR.X  R2,[R0,R1,LSL#2]`		2
		Total	13	16

Figure 4. Sample Thumb-2EE Code (*.X suffix indicates a new Thumb-2EE instruction)*

Summary

The popularity of managed execution environments and virtual machines across many embedded markets has been driven by the security, portability and time-to-market benefits that are associated with their deployment. Java has been particularly successful in penetrating the wireless handset market, but other applications including set-top box, digital TV and automotive are increasingly benefiting from deployment of an execution environment.

The ability to deliver a sufficient level of performance to enable execution environments to satisfy user's needs, while maintaining efficiency in code density, is key to unlocking the market potential for many high growth applications.

Jazelle DBX provides direct hardware execution for Java on ARM cores to boost performance for direct interpretation. This approach offers very efficient memory utilization within an environment where latency effects such as start-up delays and smoothness in performance are not a major factor e.g. because of high GHz processor speeds.

To complement Jazelle DBX, ARM has introduced Jazelle RCT to support the latest compilation technologies for Java and other execution environments. By ensuring that techniques such as AOT and JIT can be implemented efficiently without excessive code bloat, Jazelle RCT provides a platform for highly-efficient execution performance with Java and other VM technologies.

Together with ARM's Jazelle DBX technology, Jazelle RCT offers a roadmap to efficient implementation of execution environments in hardware on ARM platforms. Benchmark results have confirmed the effectiveness of the Jazelle RCT approach in enabling improved performance, while limiting code bloat to within 10% (on average) above the size of the original bytecode.

ARM's Jazelle family provides the ability to combine Jazelle DBX and Jazelle RCT on processor cores with mid-range performance. This will give developers the flexibility to use the optimum combination of interpretation and compilation to ensure the best performance from the execution environment within the constraints of the available hardware resource.

ARM offers class-leading solutions through its own hardware and software technology, as well as the ARM Connected Community. Together, ARM and the Connected Community partners offer high-quality execution environment solutions for the ARM Architecture that deliver the optimum combination of high performance, low power and low cost.

References:

[1] High performance Java on embedded devices.
Jazelle technology: ARM acceleration technology for the Java Platform
Chris Porthouse
ARM Ltd
http://www.arm.com/products/solutions/Jazelle.html