Scylla: A Smart Virtual Machine for Mobile Embedded Systems

Phillip Stanley-Marbell
Department of Electrical and Computer Engineering
Rutgers University
narteh@ece.rutgers.edu

Liviu Iftode
Department of Computer Science
Rutgers University
iftode@cs.rutgers.edu

Abstract

With the proliferation of wireless devices with embedded processors, there is an increasing desire to deploy applications that run transparently over the varied architectures of these devices. Virtual machines are one solution for code mobility, providing a virtualized processor architecture that is implemented over the individual node architectures. Proposed virtual machines for embedded systems are generally slow and consume significant energy, making them unsuitable for devices with limited processing power and energy resources.

Presented is a novel virtual machine architecture, Scylla, specially designed for mobile embedded systems, that is simple, fast and robust. In addition to a basic instruction set, Scylla supports inter-device communication, power management and error recovery. To make on-the-fly compilation extremely efficient, the instruction set closely matches popular processor architectures that can be found in embedded systems today. This paper describes Scylla, along with a preliminary evaluation of its performance, including the costs of the on-the-fly compilation and the overhead of having a virtual machine, based on simulations and measurements on a prototype system.

1 Introduction

Mobile devices with embedded processors are playing an ever increasing role in our daily lives. There are large numbers of devices with embedded intelligence, ranging from personal digital assistants and cellular phones to environment monitoring sensors. An increasing number of these devices are networked and can interchange data with each other on the global Internet. The main benefits of these networked devices lie not in their computational capability, but in the unique features each device might have, such as its geographical location, sensors, communication capability and energy resources. However, the processor architectures of these devices encompass a broad range, making the interchange of data and applications between devices difficult.

To overcome the problem of wide variation in processor architectures, one solution is to define a virtual machine [7, 17] which is implemented over the different processor architectures, shielding applications from the details of the underlying hardware and thus facilitating code and data mobility. Another approach is to provide a means of performing remote processing [11], permitting applications to initiate computation or request data on a remote device. Remote processing generally lacks flexibility, limiting applications to a fixed set of high level operations, and has the added cost of communication with remote servers. Virtual machines on the other hand attempt to provide a complete programmable machine, that is independent of the underlying processor architecture, and usually provide a complete set of low level primitives.

For a large majority of embedded devices, energy is limited, and the execution environment should provide some means of logging and preventing the excessive use of energy resources by applications. For a virtual machine architecture, energy consumption can be controlled at several points, including the host operating system over which the virtual machine runs, the compilation of byte-code into native machine code if downloaded code is on-the-fly compiled, and the execution of downloaded code. Contemporary virtual machines such as [12, 7, 17] provide neither an implementation optimized for low power consumption nor a means of power management.

This paper describes Scylla, a virtual machine architecture specially designed for mobile embedded systems. In addition to a simple, yet comprehensive instruction set that can be easily compiled on-the-fly if necessary, the virtual machine supports inter-device communication, power management and error recovery. Communication primitives enable applications running over the virtual machine to communicate with other devices. Explicitly exposing the action of communication to the virtual machine enables estimation of the amount of communication an application will perform. Applications run over Scylla may designate code to be executed if exceptional conditions occur, permitting

0-7695-0816-2/00 $10.00 © 2000 IEEE

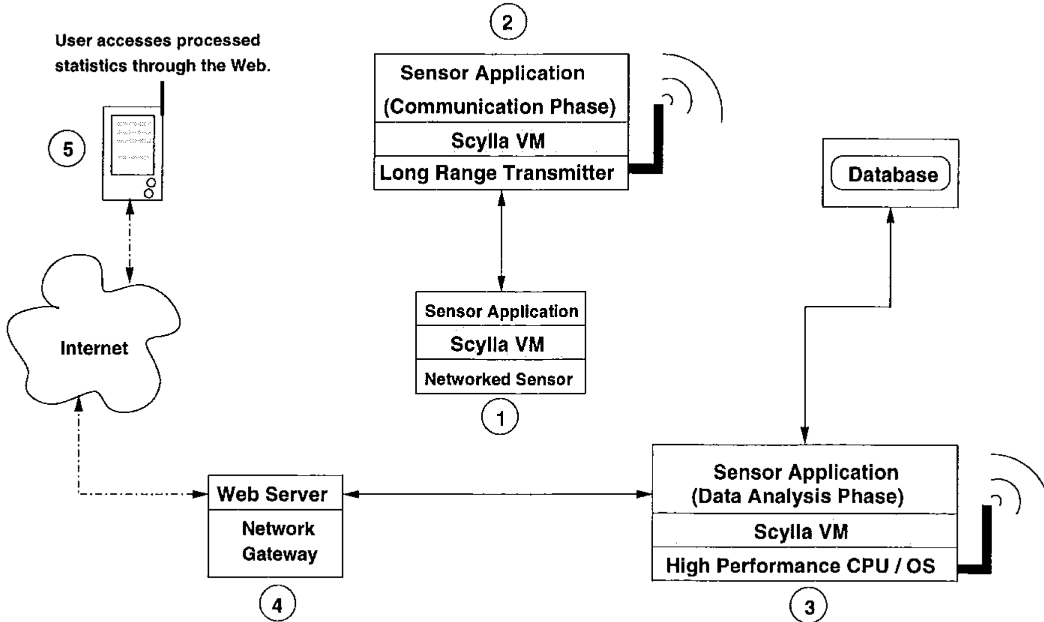

Figure 1. An Example Application

the graceful recovery from errors. If an application is accepted for execution, the virtual machine guarantees that it will have enough resources to execute the provided exception handling code. Power management is performed both actively and passively. Applications may provide the virtual machine with an estimate of their dynamic instruction count, hence implied computational energy usage, or this estimation may be performed during on-the-fly compilation. Applications that are deemed to be expensive in terms of energy, can be terminated gracefully by executing the application provided error handling code. The virtual machine was designed to closely match the architectures of most modern microprocessors and microcontrollers, making register allocation and on-the-fly compilation trivial and inexpensive.

The target class of applications for Scylla is small networked devices such as networked sensors [4, 2, 5], which have constrained computing resources, limited energy resources, and have communication interfaces. Such devices will benefit greatly from being able to safely execute mobile code, permitting applications to be deployed in large networks, without having to be hard-coded into each sensor at the time of deployment.

The contributions of this work include a novel virtual machine architecture with integrated facilities for computation and communication, power and resource management and graceful error recovery. The rest of this paper is organized as follows. The next section provides a motivating example for our work, section 3 describes related work, section 4 details the architecture of Scylla, section 5 discusses provisions for the safe execution of foreign code, section 6 discusses preliminary performance data and we conclude with section 7.

2 Example

An intelligent sensor has been deployed in a metropolitan area, to gather statistics on traffic congestion. The device's activities include monitoring automotive and pedestrian traffic, logging the information to a database, and providing real-time access to statistics through a Web interface. Since a large amount of data is collected, and the interpretation of this data is specific to the context of this sensor, the application running on the sensor must perform all data analysis and formatting, prior to supplying it to the the outside world through the Web interface. The sensor may communicate with other sensors, transmitters, network gateways, workstations and servers through standardized protocols, but these other devices have no knowledge of the the structure of the data being gathered by the sensor. Such a device must be able to endure extended deployment times, without a permanent source of power. It might obtain a majority of its energy via solar power or mechanical vibration [8], but must still function correctly and reliably in bad weather and at night. These requirements dictate that the device conserve and manage its resources as best as possible.

Figure 1 illustrates a typical scenario for the intelligent sensor. The sensor application originally resides on the sensor device with limited energy resources, and gathers statistics from its environment, such as counts of passing automobiles, ambient temperature, pressure, or levels of acoustic activity. The sensor application runs over a Scylla virtual machine and may communicate with other devices in its immediate vicinity. Due to limited energy resources, it may only transmit data over short ranges. To overcome this limitation, the application migrates to another device with a more powerful transmitter, from which it migrates yet once more to a high end server, also equipped with a Scylla virtual machine. There, it performs detailed analysis of the gathered data (which migrated with it), stores a copy in a database, then generates and uploads a web page with the updated traffic statistics to a web server. A commuter curious about the driving conditions in that metropolitan area accesses the web server, and sees up-to-date predictions for traffic delays.

The same infrastructure (sensor, servers, communication network) could simultaneously be used for an entirely different application, such as tracking pedestrian traffic or monitoring weather conditions, since there is no hard link between applications and the hardware. The key to this flexibility is to be able to safely execute and exchange code between devices with different architectures and hardware capabilities.

3 Related Work

Systems using virtual machines with interpretation or on-the-fly/just-in-time compilation, exemplified by the Inferno operating system's Dis virtual machine [17], the Java Virtual Machine [7], Omniware [12] and dating back to the IBM Virtual Machine System/370 [9], facilitate compatibility between different architectures by providing a uniform virtualized view of the underlying hardware. The abstractions provided by the virtual machine vary, and are tied to the intended model of usage.

The virtual machine system described in [9] served the dual purpose of protecting users of a time-shared system from one another and providing backward compatibility across generations of the IBM S/360 and its successors. The Java virtual machine [7, 3] was originally intended for long-lived reliable systems, providing platform independence across heterogeneous networks. Current Java virtual machine implementations are slow and memory hungry, making them inappropriate for real-time embedded systems [16]. The issue of performance has been addressed with the provision of *just-in-time* (JIT) compilers and *continuous compilation* [6, 10]. JIT compilers add large startup latencies, a problem which is addressed by continuous-compilation systems such as the Java Hotspot compiler [6]. The distributed virtual machine described in [13], factors out components of the Java virtual machine to enable distribution of tasks traditionally performed by a single node. The Java 2 Micro Edition's K virtual machine [14] provides a version of the Java virtual machine targeted at consumer electronics. Even though small compared to other Java virtual machine implementations, it does not specifically address issues such as energy efficiency and error recovery which are critical in small networked devices. The Dis virtual machine [17] was designed specifically for on-the-fly compilation rather than interpretation. Dis has complex instructions for performing operations such as creating processes, performing operations on lists and communicating between processes running over the virtual machine. The communication facilities provided by Dis only permit communication between two processes running over the same virtual machine. The Omniware virtual machine described in [12] is similar to Scylla in terms of the instruction set architecture. In [12], the instruction set was designed by analyzing dynamic instruction usages in contemporary RISC and CISC architectures. It has a RISC architecture with a number of complex instructions to perform operations such as memory-to-memory block moves. High level instructions were only provided for instructions which showed significant use in the dynamic instruction traces of the systems investigated by the authors.

The VCODE dynamic code generation system described in [1] uses a representation that is an idealized RISC architecture. VCODE performs code generation *in-place*, reducing the memory and computational cost of code generation, and generates fast native code. The architecture of Scylla follows this principle, and by closely matching the architecture to common RISC architectures, just-in-time compilation is reduced to a trivial re-mapping.

4 Architecture

Figure 2 illustrates the overall structure of a device hosting Scylla. A typical system consists of the hardware device (processor, memory, peripherals), an optional operating system, the Scylla virtual machine, and the applications that run over it. The virtual machine may interact directly with the hardware, may interface to it through system calls to an operating system, or a combination of both. It provides a uniform interface to all applications that is independent of the underlying hardware and is responsible for accepting and executing applications, subject to checks to ensure that they do not violate security policies and energy usage constraints of the device. The operating system and communication protocol issues are not discussed in this paper.

Applications run over Scylla are structured into entities called *modules*. The components of a module are logically rather than physically grouped i.e., they are not in the same

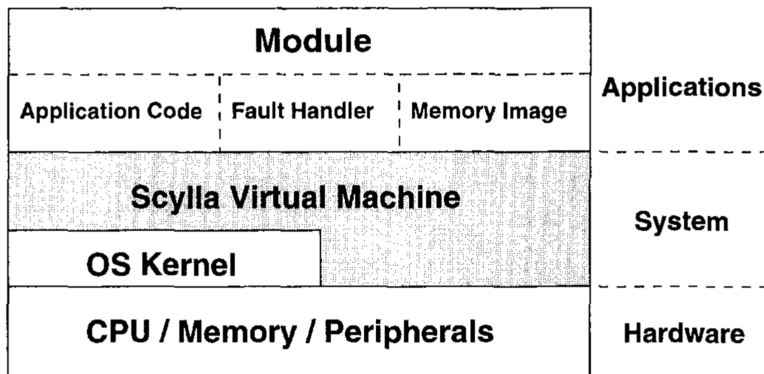

Figure 2. Typical structure of a host running Scylla

binary image, but rather separate pieces that may be provided by different sources. A module consists of application code, a memory image, and a fault handler. The application code is the only mandatory component of a module, and typically contains the code to be executed by the virtual machine when the module is loaded. The memory image component contains data to be loaded into memory before execution. If a module migrates from one device to another, the memory image, which may contain modified data, may migrate with it, thus the state of an application may be maintained across different devices. The fault handler contains code that the application has designated as operations to be performed if exceptional conditions arise during it's execution.

The application code component contains Scylla bytecode preceded by a header containing the number instructions in the component, an estimate of the energy resources that will be consumed on execution (in the form of a predicted dynamic instruction usage), a digital signature and other miscellaneous information. The digital signature may be checked by the virtual machine to assign a level of trust to the module as a whole or to the energy consumption estimate.

The memory image component contains data to be loaded into memory prior to the execution of the module. It is preceded by a header that specifies the number of bytes of data in the image, and the number of padding zero bytes to be appended to the data before loading into memory. Scylla does not permit dynamic memory allocation in the traditional sense, but modules may allocate memory upon receipt by the virtual machine by specifying a non-zero number of padding bytes.

The structure of the fault handler component is identical to that of the application code component. In addition, there are several restrictions on the instructions that may be present in the fault handler. The fault handler may not contain any backward control transfer, or memory access instructions. Forbidding backward control transfer in the fault handler permits bounding of the estimated dynamic instruction usage and hence the energy consumption of the fault handler. This permits Scylla to guarantee the execution of at least the module's fault handler, once the application has been accepted.

4.1 Module Loading

A client first sends information about the module to the virtual machine, based on which the virtual machine decides whether to continue the negotiation necessary to accept the module. The initial information sent may include an instruction count and hence an implementation dependent estimation of the energy required to execute the module, resources such as amount of memory necessary for successful compilation and execution of the module or the presence of specific hardware resources such as sensors. It then sends one component at a time (fault handler, application code, memory image), each preceded by a checksum, to allow the virtual machine to verify whether it already has the component cached, preventing unnecessary transmissions.

On receipt of all the components of a module, the virtual machine compiles the fault handler, then decides whether or not to compile the application code. The compilation of the fault handler enables the virtual machine to determine if it contains any illegal instructions and also to estimate its run-time energy cost. The cost of compilation is small, making this approach more attractive than inspecting the code, then later on having to interpret it instruction by instruction. If compilation of the fault handler fails for any of the above reasons, further processing on the module is aborted. The virtual machine next compiles the application code and attempts to run it. If any illegal instructions or memory bounds violations are detected during compilation or execution of the application code, execution of the fault handler is initiated. Synchronous exceptions cannot occur

during the execution of fault handler, due to the restrictions placed on the permitted instructions in fault handlers.

4.2 Module Energy Estimation

Energy estimation is performed for the energy consumed by the processor and communication interfaces only. This methodology is appropriate for systems in which the energy usage by the processor and communication interfaces dominate the overall system energy usage, as is typical of devices such as small wireless network sensors, which have no display or disk. A benefit of having communication primitives as part of the virtual machine instruction set, is the ability to obtain a crude estimate of how much communication will be performed by an application, and hence estimate how much energy will be expended in communicating, while the application is active.

The virtual machine estimates the energy cost of application code and fault handlers via a simple table lookup while compiling instructions. The entries in the table correspond to energy estimates obtained for the native architecture via instruction level power analysis similar to that in [15]. In general this energy prediction is inaccurate due to the fact that there is no easy way to statically determine the dynamic instruction usage. The virtual machine will terminate applications whose estimated energy cost exceeds some limit (i.e. the one agreed upon at the module loading time), and attempt to execute the application's fault handler. Since the fault handler is restricted to not having any backward control flow instructions, its dynamic instruction usage will always be less than or equal to its static instruction usage, and energy estimation during compilation of the handler is more accurate. The acceptance of a module by the virtual machine is contingent on the energy cost estimation of the module's fault handler. Once a module is accepted, the virtual machine provides a guarantee that should the need arise to execute the fault handler, there will be sufficient energy resources to complete its execution. Modules are thus assured that once accepted for execution, even if their application code is rejected due to energy cost or illegal instruction behavior, their fault handlers will be executed, permitting graceful error recovery.

In general, the energy required to execute a module may be dominated by the cost of communication. It is possible to bound these costs based on a knowledge of the transmission medium, amount of data expected to be transmitted or received, and a knowledge of the communication protocols involved. The current implementation of Scylla assumes a fixed cost per payload byte to be transmitted, and estimates the energy cost based on the payload size of the communication. We are investigating the impact of the aforementioned issues on the power management within the virtual machine, however, a detailed discussion is beyond the scope of this paper.

4.3 Error Recovery

Error recovery is facilitated through the module's fault handler. Control is irrevocably transferred to a fault handler either if exceptional conditions such as low battery levels on a battery powered device or synchronous or asynchronous machine exceptions relating to the execution of the application code occur, or if the application executes a `fault` instruction, explicitly requesting that control be passed irreversibly to its fault handler. A handler must not contain any potentially excepting instructions, such as memory loads and stores, must not contain any `fault` instructions and must not contain any backward control transfer instructions. Additionally, there is an implementation dependent limit placed on the size of the fault handler. The last two restrictions permit the accurate estimation and bounding of the energy cost of the handler.

There is no restriction placed on what the job of the fault handler must be, and it may be used to implement any general or application specific event handling that must be performed upon application termination. The fault handler will typically be used for "cleaning up" after an application exits. It may be used to migrate a module that has failed due to resource constraints on the local system, to another device, or may be used by applications processing sensitive data to clear out the contents of their memory image, either in the event of abrupt termination or at successful completion. The virtual machine guarantees that once a module is accepted for execution, there will always be sufficient system resources to execute its fault handler, and the fault handler will execute to completion.

4.4 Instruction Set

The virtual machine has sixteen 32-bit general purpose registers, R0 through R15. It is a load-store architecture, with all operations being performed on registers. The instruction set provides abstractions for performing operations like addition, negation, shifts, memory access, control transfer, and three high level instructions, `fault`, `xin` and `xout`. None of the instructions are explicitly typed. The `fault` instruction transfers control to the program's fault handler based on the values of its register operands. The `xin` and `xout` instructions provide primitives for communication. They take as operands a start memory address, number of bytes to be transmitted, port to use and the 128 bit destination address, broken up into four 32-bit fields denoted `s`, `n`, `i` and `addr!addr!addr!addr` respectively. The port parameter identifies which medium the communication should be performed on, and dictates the semantics of the address parameter. The port specified

may correspond to local memory, in which case the `xin` or `xout` instruction will function as a block memory move, or it may specify a medium for communicating with other devices, with the semantics of the address being dictated by that medium.

Instructions are of variable length, for best code density. Fixed length instructions would mandate instructions which needed fewer bits than the fixed instruction length to be padded up to the instruction length, which would in turn need to be the minimum required to encode the instruction with the most information. Instruction set architectures to be implemented in hardware generally benefit from fixed length instructions, since this removes the restriction of having to serialize the fetch and decode of subsequent instructions. Since the instructions in Scylla byte-code are just an intermediate representation that will get re-mapped to one or more instructions on a hardware architecture, we can use variable length instructions without any loss in application performance.

Table 1 compares the core of Scylla's instruction set with the PowerPC, ARM7 and Hitachi SuperH RISC architectures. These architectures are extremely popular in embedded systems such as personal digital assistants (PDAs), digital cameras and mobile phones. As can be seen from the table, most instructions in the core of Scylla's instruction set map directly to instructions in the architectures compared. Though not shown, the cases that do not map directly can be replaced with short sequences of 2 – 5 instructions. The architectures compared in Table 1 are typical of contemporary RISC architectures. CISC architectures should not pose a problem, since most support a superset of the operations provided by RISC processors, and support memory operands to instructions. Thus, even though most CISC architectures generally have a limited number of registers, the larger register files of RISC processors may be emulated by an in-memory register file.

Example

Figure 3 highlights a few features of the architecture with an example, a data logging application, implementing the pseudo-code below.

```
Initialize counters.
while (data collected < 1MB)
{
   Read in 1KB from sensor (device D1).
   Avg. into a 32-bit word, store in memory.
   Increment memory address for stores.
}
Transmit 4KB of averaged data to device D2.
```

The code in Figure 3 depicts the assembler mnemonic form, prior to any necessary checks in the form of `fault` instructions being inserted before the `ld` and `st` instructions. In this particular application, analysis of the code would reveal that no `fault` instructions need to be inserted, since all memory references are statically unambiguous. The application repeatedly collects 1KB of data from a peripheral on port `D1`, averages it, and stores the result in memory. After collecting 1MB worth of raw data, it transmits the 4KB worth of averaged data to a database, which is connected to the medium on port `D2` and has address `0xf!0xf!0xe!0xc`. The raw data is read into memory starting at address `0x0` and the averaged data is written into memory starting at address `0x800`. The application compiles to 113 bytes of Scylla byte-code.

```
        /*      Reset counters  */
        ANDI    #0x0,           R0
        ANDI    #0x0,           R1
        ANDI    #0x0,           R4
        ANDI    #0x0,           R10
        ADDI    #0x1,           R1
        ADDI    #0xa,           R0
        SHLLN   R0,             R1
        ADD     R1,             R4
        ADDI    #-4,            R4
        ADD     R1,             R10
        ADD     R10,            R10

        /*      Read in data    */
L0:     XIN     0x0,            0x400,   D1,   0x0!0x0!0x0!0x0
        ANDI    #0x0,           R0
        ANDI    #0x0,           R3
        /*      Average data    */
L1:     LD      @R0,            R2
        ADD     R2,             R3
        ADDI    #0x4,           R0
        BLE     R0,             R4,      L1
        ANDI    #0x0,           R0
        ADDI    #0xa,           R0
        SHRLN   R0,             R3
        ST      @R10,           R3
        ADDI    #4,             R10
        /*      Got 1Mbyte ?    */
        ADDI    #-1,            R1
        ANDI    #0x0000,        R0
        BLE     R0,             R1,      L0

        /*      Send out data   */
        XOUT    0x800,          0x1000,  D2,   0xf!0xf!0xe!0xc
```

Figure 3. A data logging application.

4.5 Register Allocation

All register allocation is done statically, when the module is created, making the job of the on-the-fly compiler much easier, and permitting low startup latencies for on-the-fly compiled code. This is possible because we know *exactly* how many registers are available, at compile time. Register allocation is not trivial, and is an issue that hinders performance of just-in-time compiled code on Inferno's Dis virtual machine [17]. The fixed register set does however lead to interesting issues. Even though there is a significant similarity between the instruction sets of contemporary RISC architectures, these architectures generally fall into two distinct classes, with respect to register operands to instructions. Most architectures such as the PowerPC, SPARC, ARM and MIPS use three address instructions, and explicitly name a separate destination register, whereas some architectures such as the Hitachi SuperH RISC archi-

Scylla	Hitachi SuperH	PowerPC	ARM 7
ADD Rm, Rn	ADD Rm, Rn	ADD Rn, Rn, Rm	ADD Rn, Rm, Rn
AND Rm, Rn	AND Rm, Rn	AND Rn, Rn, Rm	AND Rn, Rm, Rn
OR Rm, Rn	OR Rm, Rn	OR Rn, Rn, Rm	ORR Rn, Rm, Rn
XOR Rm, Rn	XOR Rm, Rn	XOR Rn, Rn, Rm	EOR Rn, Rm, Rn
ADDI #IMM, Rn	ADD #IMM, Rn	ADDI Rn, Rn, #IMM	ADD Rn, Rn, #IMM
ANDI #IMM, Rn	N/A	ANDI Rn, Rn, #IMM	AND Rn, Rn, #IMM
ORI #IMM, Rn	N/A	ORI Rn, Rn, #IMM	OR Rn, Rn, #IMM
XORI #IMM, Rn	N/A	XORI Rn, Rn, #IMM	EOR Rn, Rn, #IMM
BRA DISP	BRA DISP	BL DISP	BAL DISP
LD @Rm, Rn	MOV.L @Rm, Rn	LD Rn, 0(Rm)	LD Rn, [Rm, #0]
ST @Rm, Rn	MOV.L Rn, @Rm	STD Rn, 0(Rm)	STR Rn, [Rm, #0]
NOT Rm, Rn	NOT Rm, Rn	N/A	MOVN Rn, Rm
NEG Rm, Rn	NEG Rm, Rn	NEG Rn, Rm	SUB Rn, Rm, #0

Table 1. Comparison of Scylla, ARM 7, PowerPC and Hitachi SuperH Architectures.

tecture use two address instructions, with an implicit destination register. Scylla employs two address instructions with an implicit destination register.

If the virtual machine architecture used explicit destination registers, the cost of transforming the instruction stream to match another architecture that also used explicit destination registers would be nil, but for architectures that used implicit destination registers, three additional instructions would have to be generated for each virtual machine instruction, and an additional register would be required to hold temporary values. For example, to implement a register add on a two address architecture if the virtual machine used three address instructions, the native instruction sequence needed would be:

```
MOV    Rn,    Rtmp
ADD    Rm,    Rn
MOV    Rn,    Rd
MOV    Rtmp,  Rn
```

However if the virtual machine architecture uses implicit destination registers, mapping instructions to like architectures is simple, as is transforming from implicit destination register to explicit destination register formats.

4.6 Compilation to Native Architectures

The instruction set architecture of the virtual machine is well suited for on-the-fly compilation. Because most Scylla instructions map directly to instructions in the instruction sets of modern microprocessors and microcontrollers, compilation is usually just a matter of re-encoding an instruction word. Figure 4 illustrates the compilation of an ADD R6, R13 instruction from Scylla byte-code to an instruction on the PowerPC architecture. The ADD instruction in the Scylla architecture has opcode 0, and for this

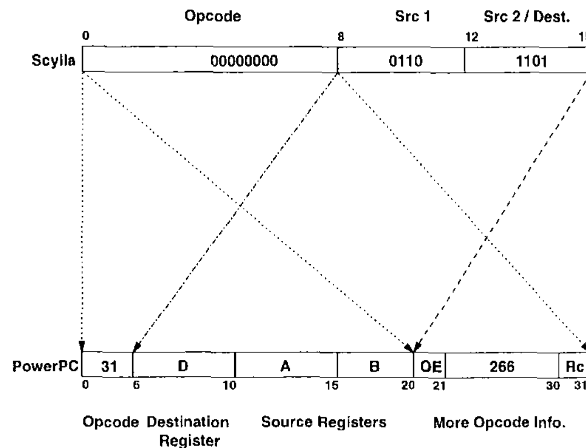

Figure 4. Compiling an ADD R6, R13 instruction from Scylla byte-code on the PowerPC architecture.

example, the source and source/destination fields, are set to 6 and 13 respectively. In the PowerPC instruction in the lower portion of Figure 4, the shaded fields contain opcode information, (in this case indicating an ADD instruction) and remain fixed irrespective of the register operands. The OE and Rc fields specify which variant of the ADD instruction this is, and for this example will both be set to 0.

The PowerPC uses explicit source (fields A and B in the figure) and destination register (field D) operands, thus source field of the Scylla instruction is placed in the A field, and the source/destination field of the Scylla instruction is placed in both fields B and D. Thus, the PowerPC instruction corresponding to the ADD R6, R13 Scylla instruction can be constructed with just 4 operations, using C syn-

tax:

```
/*  ppc_instr will contain the  */
/*  assembled PowerPC instr.    */
ppc_instr = 0x7C000214;
ppc_instr |= (src2<<6);
ppc_instr |= (src2<<15);
ppc_instr |= (src1<<10);
```

The current implementation of the on-the-fly compiler is rather dimunitive, compiling to 1.1KB of object code for the Hitachi SuperH RISC architecture.

5 Running Foreign Code Safely

The ability to safely run foreign code is a major concern, and even more so in embedded systems, since many embedded processors do not have memory protection hardware such as memory management units (MMUs). The primary issue addressed in this section is illegal memory accesses by applications. Memory access checks are implemented through binary rewriting on the original byte-code, before compilation. Memory accesses that cannot be disambiguated statically are guarded by inserting a `fault` instruction to check the address of the memory access, and if out of bounds, transfer control to the application's fault handler. As an example, the following Scylla code will cause control to be transferred to the module's fault handler if the address of the memory load is greater than or equal to the value in register R5.

```
FAULT   R10,    R5
LD      @R10,   R8
```

Not all memory accesses need to be guarded, and checks are only necessary for register indirect loads and stores for which the addresses cannot be statically disambiguated, reducing the explosion in code size and performance degradation that would result from the insertion of numerous checks for all memory accesses. Performing the insertion of memory access checks at the byte-code level enables the checking to be done in an architecturally independent manner, possibly on a device other than the one that will eventually run the code.

A distributed architecture such as that described in [13] may be constructed out of several Scylla nodes with different architectures and different processing capabilities. The static checking might then be relegated to one or more trusted nodes, which would digitally sign each module component upon completion of the analysis. Digital signatures for application binaries have been employed in other virtual machine architectures such as those described in [17, 3] to provide a degree of assurance as to the integrity of the application and the authenticity of its origin. Architectures without MMUs may ignore the authenticity of signed modules if

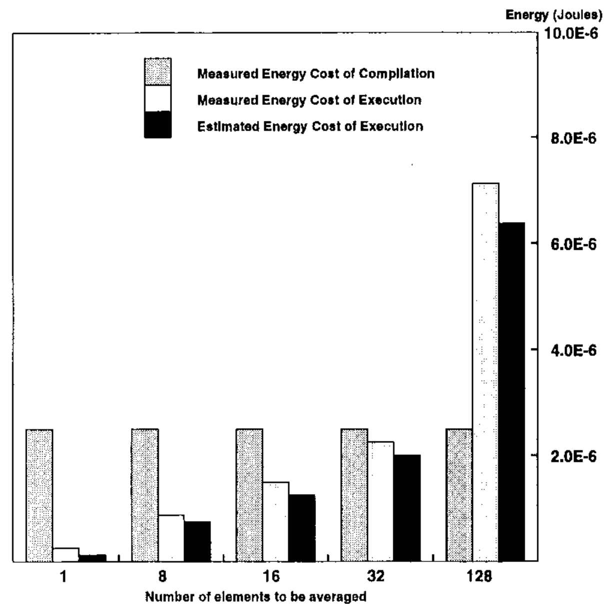

Figure 5. Relative cost of on-the-fly compilation.

they have been marked as already statically checked, since there is the risk that the apparently benign module might originate from a malicious node that has "faked" the signature of a trusted entity, or the module might be from a trusted entity that has been compromised or is operating incorrectly. If the static checking is done on a node that is unconstrained in terms of resources, an minimal number of checks may be inserted by performing detailed code analysis.

There is no way for programs run on the virtual machine to transfer control to arbitrary PC locations (register indirect jumps are not provided in the architecture), thus there is no way for applications to transfer control to code that has not been checked for illegal accesses. Register indirect jumps are not supported because even though it can be enforced that the destination of the jump has to be in valid memory, in the dynamic instruction stream, jumps could be used to circumvent the memory check instructions used to protect memory accesses.

6 Performance

Scylla currently runs on the SH-3 LCEVB, a processor evaluation board for the Hitachi SH-3 microcontroller. In the following analysis, hardware measurements were augmented with instruction usage statistics obtained from an architectural simulator, which models the processor, cache,

Scylla Instruction	Compilation Cost (instructions)	Execution Cost (instructions)	Measured Compilation Cost(Joules)
ADD	12	1	177.1E-9
ADDI	22	1	321.5E-9
BLE	19	2	275.E-9
BRA	6	2	86.9E-9
MULT	27	3	386.2E-9
SHLLN	18	1	260.9E-9
XOUT	28	N/A	404.6E-9

Table 2. Code generation cost statistics on the Hitachi SH3 LCEVB.

```
L1:     LD      @R0,    R2
        ADD     R2,     R3
        ADDI    #0x4,   R0
        BLE     R0,     R4,     L1
        ANDI    #0x0,   R0
        ADDI    #0xa,   R0
        SHRLN   R0,     R3
        ST      @R10,   R3
        ADDI    #4,     R10
```

Figure 6. A kernel from the data logging application of Figure 3.

memory and serial communications controller, and is used to obtain statistics such as dynamic instruction count, CPU cycles for execution and estimates of circuit switching activity. Energy costs are determined by first running the application over the simulator to obtain an accurate cycle count, then measuring the average current draw on the development board, for the same application executing repeatedly. The measurements are for the entire board, including the processor, memory and peripherals. An assembler to convert Scylla assembler to the binary byte-code format is available, and is also capable of generating C array definitions from the assembler. These can then be included from the source of the virtual machine implementation, making it possible co-simulate a Scylla application and the virtual machine on the architectural simulator.

The relative cost of on-the-fly compilation was investigated, to determine whether or not compilation was worth the effort, and at what point the cost of compilation exceeded the cost of execution. Figure 6 shows the innermost loop from the example in Figure 3. It computes the average of the elements of an array stored in memory, and stores the result to another memory region. The particular code shown is specifically for an array of 1024 elements. The kernel was run over the virtual machine on the SH-3 LCEVB, and energy consumption measurements taken for averaging array sizes of 1, 8, 16, 32 and 128 elements. For all these runs, the measured cost of compilation from byte-code to native machine code remained approximately the same, since the the only variation was the constant defining the array size (not shown in the figure). The relative costs of compilation and execution for the 5 runs is shown in Figure 5. As can be seen from the figure, there is a breakpoint around array sizes of 32 for which the cost of compilation begins to play a smaller role in the overall cost of executing a module. As can be inferred from Figure 6, this variation is due to the increasing dynamic instruction usage for computing the averages of the larger datasets. It is not easily determined by inspection whether the cost of compiling a module will be insignificant compared to the cost of its execution, but even for small portions of code as in this example, averaging a 128 element array, the cost of compilation is likely to be insignificant compared to the cost of execution.

Figure 5 also shows the estimated cost of executing the kernel. The estimate is obtained by running the implementation of Scylla, compiled for the SH3 processor, over the architectural simulator. The code for the kernel is statically included in the virtual machine, which is modified to directly compile and run it. For each SH-3 instruction that is encountered, the simulator looks up the energy cost of executing the instruction from a table of measured values.

Even though the cost of compilation relative to the overall average cost of execution will generally not be significant, the cost of the execution is itself also small. Table 2 shows the average costs of compilation and execution of a sampling of Scylla instructions. The average cost of compilation of a module per Scylla instruction is 19 instructions on the SH-3 LCEVB, consuming an average of 273.2E-9 Joules. Most Scylla instructions map directly to instructions in the native architecture, and since the register set of the virtual machine generally matches that of the actual hardware, most instructions can execute in one native instruction, as shown in the table. In Table 2, instructions such as `ble`, `bra` and `mult` that map onto more than one native instruction do not use any additional registers, and any necessary swapping of register values is accomplished

without using temporaries. For architectures that have more than 16 general purpose registers, these may be utilized by the implementation to make such swapping more efficient. The `xout` instruction initiates execution of a subroutine to perform communication, the energy cost of which is highly dependent on the communication medium.

7 Conclusion and Future Work

Presented in this paper is Scylla, a smart yet simple virtual machine architecture for mobile embedded systems, along with a preliminary evaluation of its performance. The key features of Scylla are very small footprint, energy monitoring and management, graceful error recovery and integrated communication primitives in the instruction set. The last feature is key in enabling a system to estimate the amount of communication that an application might undertake, an important issue when there is a high energy cost per bit of transmitted data.

Scylla can perform a crude estimation of the energy cost of a module upon receipt. The virtual machine guarantees that once a module is accepted for execution, there will always be sufficient system resources to, at the very least, execute its fault handler to completion. The virtual machine architecture is closely matched to that of popular processors used in embedded systems. As a result, the cost of compilation from byte-code to native machine code, in terms of native instructions and complexity of the compilation process is very small, and significant application performance can be achieved.

We have implemented Scylla on the SH-3 LCEVB, a processor evaluation board for the Hitachi SH-3 microcontroller. The current virtual machine implementation uses an on-the-fly compiler that is only 1.1KB in size. An evaluation using a synthetic application showed that the energy cost of on-the-fly compilation is insignificant compared to the cost of execution even for moderately small input data sets, and the cost of execution is in itself very small. We are currently investigating the benefits of proactively putting the processor and peripherals into low power modes. Most modern microprocessors and microcontrollers support one or more such modes, and it would be prudent to take advantage of this facility. Missing from our current methodology is the ability to accurately estimate the cost of communication. We are looking further into performing more detailed communication and transmission protocol energy analyses, for specific communication technologies.

References

[1] D. R. Engler. VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System. In *Proceedings of the ACM SIGPLAN '96 conference on Programming Language Design and Implementation*, pages 160–170, 1996.

[2] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the fifth annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 263–270, 1999.

[3] J. Gosling. Java Intermediate Bytecodes. In *ACM SIGLAN workshop on Intermediate Representations*, pages 111–118, 1995.

[4] W. R. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive Protocols for Information Dissemination in Wireless Sensor Networks. In *Proceedings of the fifth annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 174–185, 1999.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.

[6] U. Holzle, L. Bak, S. Grarup, R. Griesemer, and S. Mitrovic. Java [tm] On Steroids: Sun's High-Performance Java Implementation. In *Hot Chips 9 Symposium*, August 1997.

[7] T. Lindholm and F. Yellin. *The Java[tm] Virtual Machine Specification, Second Edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, May 1999.

[8] S. Meninger, J. O. Mur-Miranda, R. Amirtharajah, A. Chandrakasan, and J. Lang. Vibration-to-Electric Energy Conversion. In *Proceedings International Symposium on Low Power Electronics and Design*, pages 48–53, 1999.

[9] R. Meyer and L. Seawright. A Virtual Machine Time-Sharing System. *IBM Systems Journal*, 9(3):199–218, 1970.

[10] M. P. Plezbert and R. K. Cytron. Does "Just in Time" = "Better Late than Never" ? In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 120–131, 1997.

[11] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning. The Remote Processing Framework for Portable Computer Power Saving. In *ACM Symposium on Applied Computing*, pages 365–372, February 1999.

[12] R. W. S. Lucco, O. Sharp. Omniware: A Universal Substrate for Web Programming. In *Fourth International World Wide Web Conference*. MIT LCS, OSF and World Wide Web Consortium, December 1995.

[13] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. *Operating Systems Review*, 34(5):202–216, December 1999.

[14] Sun Microsystems. *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices.* Whitepaper, May 2000.

[15] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step Towards Software Power Estimation. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 384–390, August 1994.

[16] W. Webb. Embedded Java: An Uncertain Future. *Electronic Design News*, 44(10):89–96, May 1999.

[17] P. Winterbottom and R. Pike. The Design of the Inferno Virtual Machine. In *Hot Chips 9 Symposium*, August 1997.