Scylla: A Smart Virtual Machine for Mobile Embedded Systems

Phillip Stanley-Marbell
narteh@ece.rutgers.edu
Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey
Piscataway, NJ 08854

Liviu Iftode
iftode@cs.rutgers.edu
Department of Computer Science
Rutgers, The State University of New Jersey
Piscataway, NJ 08854

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. Current virtual machines are generally slow and consume significant energy, making them unsuitable for embedded 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 the processor architectures that can be found in embedded systems today. The 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 it's geographical location, sensors, communication capability and energy resources. However, the processor architectures of the devices encompass a broad range, making the interchange of data and applications between devices difficult.

To overcome the wide variation in processor architectures, the common solution is to define a virtual machine [7, 15] 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 remote procedure call or remote method invocation protocol permitting applications to initiate computation or request data on a remote device. The latter two solutions lack flexibility, limiting applications to a fixed set of high level operations, which may not represent sufficient primitives for solving arbitrary problems. 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 [10, 7, 15] 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 or even migrate in an architecturally independent manner. This communication may take place over some shared (wireless or wired) medium, or may be communication with a device attached to a processor's general purpose I/O port. Applications run over the virtual machine may designate code to be executed if exceptional conditions occur, permitting the graceful recovery from errors. Power management is performed both actively and passively. Applications may provide the virtual machine with an estimate of their expected energy usage or this estimation may

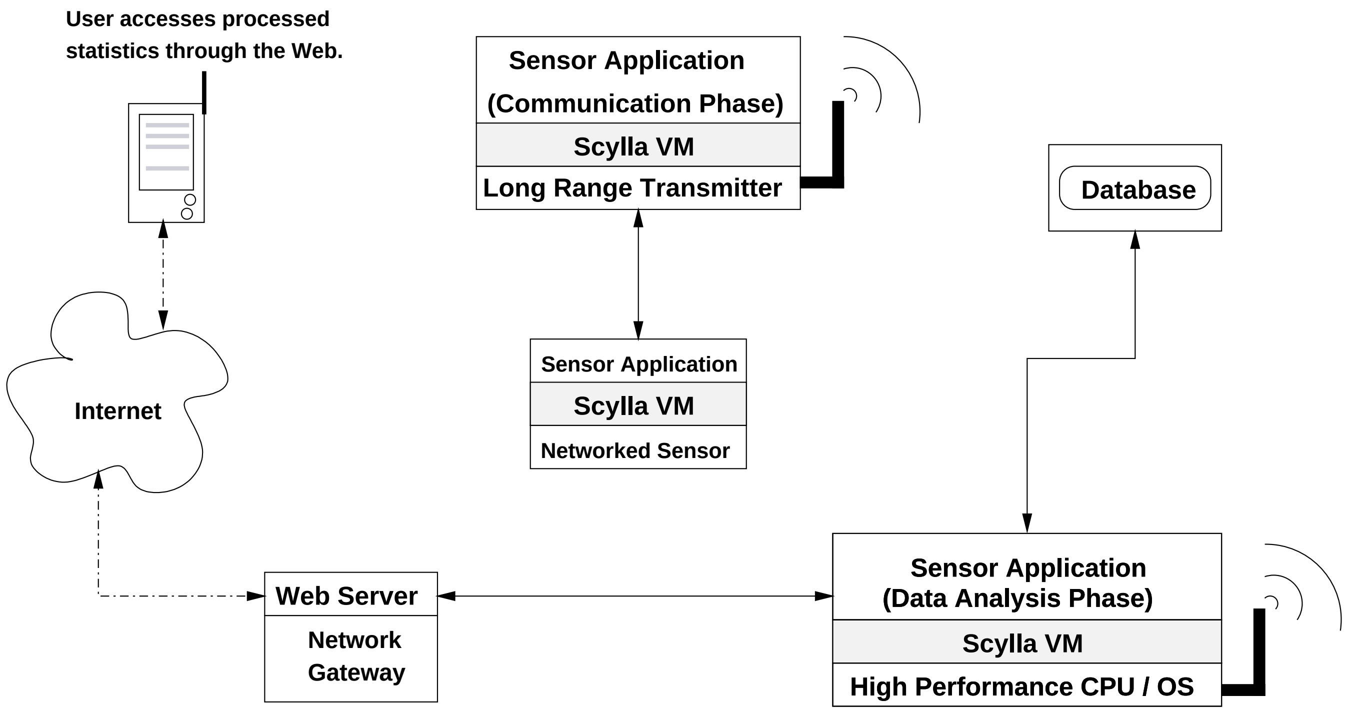

Figure 1: An Example Application

be performed during on-the-fly compilation. Applications that are deemed to be expensive in terms of energy, can be terminated gracefully through the virtual machine's error handling facility when they are loaded, during compilation or during execution. 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. Applications of Scylla include but are not limited to wired and wireless sensor networks such as those described in [5, 3], mobile communication devices and intelligent personal, auto, and household devices.

The contributions of this work include a novel virtual machine architecture with integrated computation and inter-node 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 6 discusses provisions for the safe execution of foreign code, section 7 discusses preliminary performance data and we conclude with section 8.

2. Example

An intelligent sensor has been deployed in a metropolitan area, to gather statistics on traffic congestion. The devices 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 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. Furthermore, such a device must be able to endure extended deployment times, without a permanent source of power, since it must cover a wide area and thus cannot be tethered to a fixed power supply. Such a device might obtain a majority of its energy via solar power, but must still function correctly and reliably in bad weather and at night. These requirements dictate that the device conserve and manage it's 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 it's environment. The sensor application runs over a Scylla virtual machine and may communicate with other devices in it's immediate vicinity. Due to limited energy resources, it may only transmit data over short ranges, and for short durations. 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, stores a copy in a database, and 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.

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 [15], the Java Vir-

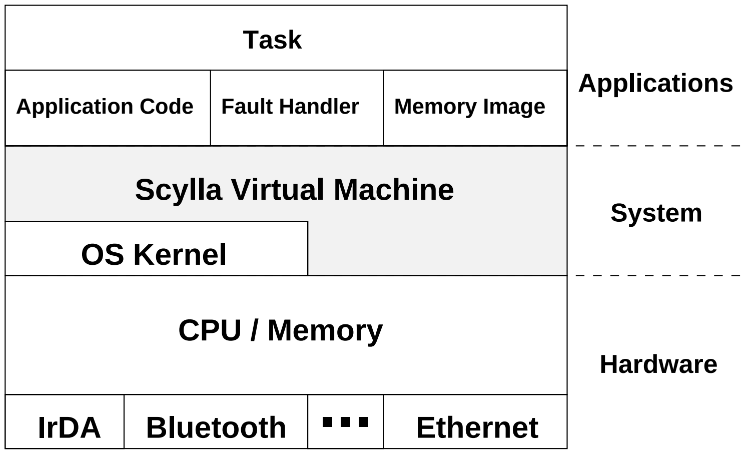

Figure 2: Typical structure of a host running Scylla

tual Machine [7] and the IBM Virtual Machine/Enterprise System Architecture [1], and dating back to the IBM Virtual Machine System/370 [8, 13], 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 [8] 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, 4] was originally intended for long-lived reliable systems, providing platform independence across heterogeneous networks. It is based on a stack architecture, has typed instructions and most instructions define simple operations. Current Java virtual machine implementations are slow and memory hungry, making them inappropriate for real-time embedded systems [14]. The speed issue has been addressed with the provision of *just-in-time* (JIT) compilers and *continuous compilation* [6, 9]. 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 [11], attempts to factor out components of the virtual machine to enable distribution of tasks traditionally performed by a single node running one or several instances of the virtual machine, into separate nodes. The Dis virtual machine [15] was designed specifically for on-the-fly compilation rather than interpretation. It has a memory-to-memory architecture, and thus looks like a conventional CISC machine, with an infinite virtual register set. Dis has complex instructions for performing operations such as creating processes, performing operations on lists and communicating between processes running over the VM. The communication facilities provided by Dis are only between two processes running over the virtual machine. A majority of the instructions provided by Dis are to support Limbo, the application programming language for Inferno. The `VCODE` dynamic code generation system described in [2] 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. The Omniware virtual machine described in [10] is similar to Scylla in terms of the instruction set architecture. In [10], 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.

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 *tasks*. The components of a task are logically rather than physically grouped i.e., they are not in the same binary image, but rather separate pieces that may be provided by different sources. A task consists of application code, a memory image, and a fault handler. The application code is the only mandatory component of a task, and typically contains the code to be executed by the virtual machine when the task is loaded. The memory image component contains data to be loaded into memory before launching the task. If a task migrates from one device to another, the memory image, which may contain modified data, will 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

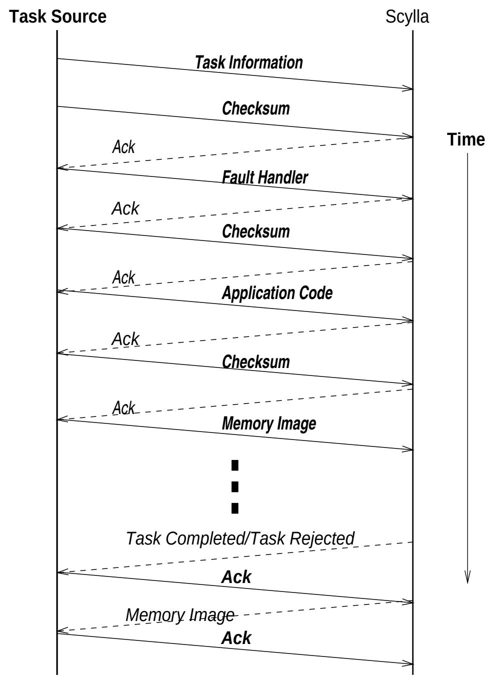

Figure 3: Receiving a Task for execution

exceptional conditions arise during it's execution.

The application code component contains Scylla byte-code preceded by a header containing the number instructions in the component, an estimate of the energy resources that will be consumed on execution, 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 task 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 task. 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 tasks 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 control transfer, or memory access instructions. Forbidding control transfer in the fault handler enables accurate estimation of the dynamic instruction usage and hence the energy consumption of the fault handler. This permits Scylla to guarantee the execution of at least the task's fault handler, once the task has been accepted.

5. Task Loading

Figure 3 shows the steps involved in sending a task to a Scylla virtual machine. The figure depicts a typical scenario in which a source of tasks sends a task to the virtual machine for execution, and upon successful completion, receives a possibly modified version of the memory image component. The source first sends information about the task to the virtual machine, based on which the virtual machine decides whether to continue the negotiation necessary to accept the task. The initial information sent may include an estimate of the energy required to run the task, resources such as amount of memory necessary for successful compilation and execution of the task or the presence of specific hardware resources such as sensors. It then sends one component at the 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 task, 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 it's run-time energy cost. The cost of compilation is relatively small, making this approach more attractive than inspecting the code, then later on having to interpret it instruction by instruction [1]. If compilation of the fault handler fails for any of the above reasons, further processing on the task 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. Upon successful completion of execution of the application code, the virtual machine may send the possibly modified memory image including any added padding bytes, to a task source or other device.

5.1 Power Management

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 [12]. 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 task loading time), and attempt to execute the application's fault handler. Since the fault handler is restricted to not having any control flow instructions, it's dynamic instruction usage equals it's static instruction usage, and energy estimation during compilation of the handler is more accurate. The acceptance of a task by the virtual machine is contingent on the energy cost estimation of the task's fault handler. Once a task 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 it's execution. Tasks 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.

[1]Compilation is performed in one pass, and the compiler occupies only 1.1K

In general, the energy required to execute a task 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.

5.2 Error Recovery

Error recovery is facilitated through the task's fault handler. Control is irrevocably transferred to a task's 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 it's fault handler. A handler must not contain any potentially excepting instructions, such as a memory loads and stores, must not contain any fault instruction and must not contain any control transfer instructions. Additionally, there is a implementation dependent size limit placed on the size of the fault handler. The last two restrictions are placed to permit the accurate estimation and bounding of the energy cost of the handler respectively.

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 task termination. The fault handler will typically be used for "cleaning up" after a task completes. It may be used to migrate a task that has failed due to resource constraints on the local system, to another device, or may be used by tasks 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 task is accepted for execution, there will always be sufficient system resources to execute it's fault handler, and the fault handler will execute to completion.

5.3 Instruction Set

The virtual machine has 16 32-bit general purpose registers, R0 through R15. It is a load-store architecture, with all operations being performed on registers. Data is retrieved from and transferred to memory by load and store instructions. Table 2 shows the complete set of instructions defined in the architecture, grouped by instruction format. The instruction set provides abstractions for performing operations like addition, negation, shifts, memory access and control transfer, and three high level instructions, fault, xin and xout. None of the instructions are explicitly typed, and all operands are treated as 32 bit data words. The fault instruction transfers control to the programs fault handler based on the values in it's 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, in Table 2. 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 the 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 performance.

Table 1 compares the core of Scylla's instruction set with the PowerPC, ARM7 and Hitachi SuperH RISC architectures, which are extremely popular in embedded systems such as personal digital assistants, 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 operation provided by RISC processors, and support memory operands to instructions. Thus, even though most CISC architectures generally have limited numbers of registers, the larger register files of RISC processors may be emulated by an in-memory register file.

Example

Figure 4 highlights a few features of the architecture with an example, a data logging application, illustrated with pseudocode below.

```
Initialize counters.

while (data collected < 1MB)
{
      Read in 1K from sensor (device D1).

      Average data read into a 32-bit word,
      store in memory.

      Increment memory address we store to.
}

Transmit 4K of averaged data to device D2.
```

The code in Figure 4 depicts the assembler mnemonic form, prior to any necessary checks in the form of fault instructions

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.

ADD Rm, Rn	Add data in register Rm to that in register Rn and store result in register Rd.
AND Rm, Rn	Logically AND data in register Rm with that in register Rn, storing result in register Rd.
OR Rm, Rn	Store logical OR of data in registers Rm and Rn, in register Rd.
XOR Rm, Rn	Exclusive OR the data in register Rm and Rn storing result in register Rd.
ADDI #IMM, Rn	Sign extend immediate IMM to 32 bits and add to data in register Rm, storing result in register Rd.
ANDI #IMM, Rn	Sign extend IMM to 32 bits and logically AND with data in Rm, storing result in Rd.
ORI #IMM, Rn	Sign extend IMM to 32 bits storing the logical OR of the result with the contents of register Rm in register Rd.
XORI #IMM, Rn	Sign extend IMM to 32 bits, OR result with the data in registerRm, and store the result in register Rd.
LD DISP, Rn	Load data from memory address of DISP (8-bit, PC-relative displacement) sign extended to 32 bits, into Rn.
ST DISP, Rn	Store data in Rn at memory address of DISP (8-bit, PC-relative displacement) sign extended to 32 bits.
BLE Rm, Rn, DISP	Conditionally branch to DISP (sign extended to 32 bits) if Rm is less than or equal to Rn.
BRA DISP	Unconditionally branch to DISP (sign extended to 32 bits).
FAULT Rm, Rn	Transfer control to fault handler if Rm is greater or equal to Rn.
LD @Rm, Rn	Store word of memory addressed by the contents of register Rm in register Rn.
ST @Rm, Rn	Store data in register Rm in the memory location addressed by the contents of register Rn.
MULT Rm, Rn, RdL, RdH	32-bit multiplication of contents of registers Rm and Rn, storing result in registers RdL and RdH.
NOT Rm, Rn	Store one's complement of data in register Rm in register Rn. This is a bitwise negation of Rm.
NEG Rm, Rn	Store two's complement of data in register Rm in register Rn, thus achieving the effect of Rn = 0 - Rm.
SHLLN Rm, Rn	Shift data in register Rn left by Rm bits, and store result in register Rn.
SHRLN Rm, Rn	Shift data in register Rn right by Rm bits, and store result in register Rn.
XIN s, n, i, addr!addr!addr!addr	Receive n bytes from 128 bit address addr on resource port i and place into memory starting at address s.
XOUT s, n, i, addr!addr!addr!addr	Send n bytes from memory starting at address s to 128 bit address addr on resource port i.

Table 2: Instruction Set

```
        /*      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 4: A data logging application.

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 1Kbyte of data from a peripheral on port D1, averages it, and stores the result in memory. After collecting 1Mbyte worth of raw data, it transmits the 4Kbytes 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. A more complex implementation might also maintain a counter in it's memory image, to keep track of how many times the application has migrated, or might log information about all the devices it has migrated through. The application compiles to 113 bytes of Scylla byte-code.

5.4 Register Allocation

All register allocation is done statically, when the task 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 easy, and is one of the issues that hinder performance in just-in-time compiled code on Inferno's Dis virtual machine [15]. 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.

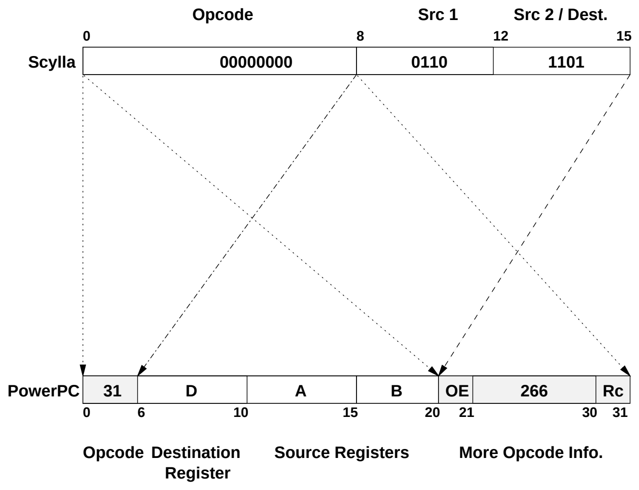

Figure 5: Compiling an ADD R6, R13 instruction from Scylla byte-code on the PowerPC architecture.

Most architectures such as the PowerPC, SPARC, ARM, MIPS use three address instructions, and explicitly name a separate destination register, whereas some architecture like the Hitachi SuperH RISC architecture use two address instructions, with an implicit destination register. Scylla uses two address instructions with an implicit destination register. Using two address instructions with an implicit destination ease compilation tremendously.

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 as shown below.

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

However if the virtual machine architecture uses implicit destination registers, the cost of mapping instructions to like architectures is nil, and the cost of transforming from implicit destination register to explicit destination register formats is also nil.

5.5 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, and the compilation is performed in one pass. Figure 5 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 example, the Src 1 and Src 2/Dest fields, are set to 6 and 13 respectively. In the PowerPC instruction in the lower portion of Figure 5, the shaded fields contain opcode information, (in this case indicate 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 src1 field of the Scylla instruction is placed in the A field, and the src2/dest 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 syntax:

```
/*                                      */
/* src1 and src2 contain Src 1 and Src2  */
/* fields of the Scylla instr. ppc_instr */
/* will contain compiled 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 compiles to 1.1KB of object code for the Hitachi SuperH RISC architecture.

6. 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 here 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 task'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, 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 [11] 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 task component upon completion of the analysis. Digital signatures for application binaries have been employed in other virtual machine architectures such as those described in [15, 4] to provide a degree of assurance as to the integrity of the application and the authenticity of it's origin. Architectures without MMUs may ignore the authenticity of signed tasks if they have been marked as already statically checked, since there is the risk that the apparently benign task might originate from a malicious node that has "faked" the signature of a trusted entity, or the task might be from a trusted entity that has been compromised or is operating incorrectly. Most nodes in such a system would therefore save the cost of performing code analysis and insertion of memory access guards. 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.

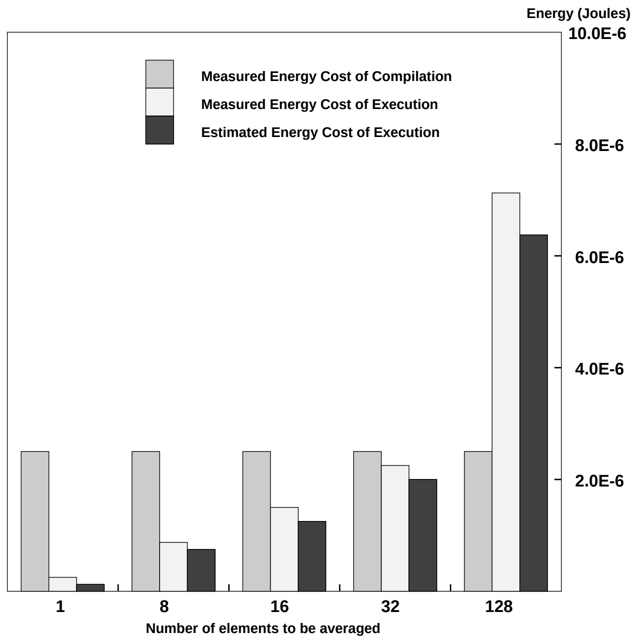

Figure 6: Relative cost of on-the-fly compilation.

```
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 7: A kernel from the data logging application of Figure 4.

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 3: Code generation cost statistics on the Hitachi SH3 LCEVB.

7. Performance

Scylla currently runs on the SH-3 LCEVB, a processor evaluation board for the Hitachi SH-3 microcontroller. The architecture independent portions have also been ported to the PowerPC 603e *Excimer* processor evaluation board and the ARM AEB-1 evaluation board. Energy measurements are performed using an off-the-shelf digital multimeter, and instruction usage statistics obtained from an architectural simulator. Energy costs are determined by running the application over the simulator to obtain an accurate cycle count, and measuring the 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. The architectural simulator models the processor, memory and serial communications controller, and is used to obtain statistics such as dynamic instruction count and CPU cycles, to augment the measurements performed on the actual hardware. 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 could then be included form the source of the virtual machine implementation, making it possible co-simulate a Scylla application and the virtual machine, without having to engage in communication to upload the task to the virtual machine.

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 7 shows the innermost loop from the example in Figure 4. It computes the average of 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 on the hardware, and energy consumption measurements taken for averaging array sizes of 1, 8, 16, 32 and 128. For all these runs, the measured cost of compilation from byte-code to native machine code remained 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 6. 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 task. As can be inferred from Figure 7, 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 task will be insignificant compared to the cost of it's execution, but even for small portions of code as in this example, averaging a 128 element array, the cost of compilation can be insignificant compared to the cost of execution.

Figure 6 also shows the estimated cost of executing the kernel. The estimate is obtained by running the same application over the architectural simulator for the SH-3. 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. Our current estimation is not very accurate, but it is evident from the figure that the estimate follows the trend of the measured values. We are currently working on improving the estimated cost through more detailed measurements and calibration. The power estimation performed by an instance of the virtual machine is separate from this estimation performed by the architectural simulator, but will benefit directly from any improvements in the energy estimation strategy and measurements, since they are implemented based on the same principles.

Even though the cost of compilation relative to the overall average cost of execution of individual Scylla byte-code instructions will generally not be significant, the cost of the execution is itself also rather small. Table 3 shows the average costs of compilation and execution of a sampling of Scylla instructions. The average cost of compilation of a task per Scylla instruction is 19 instructions on the SH-3 LCEVB, consuming an average of 273.2E-9 Joules. We believe that the the average costs of compilation can be reduced much further, since no attempt has been made to aggressively optimize our current on-the-fly compiler implementation. It is currently implemented in C and significant gains might be achieved by implementing it in assembler. Most Scylla instructions map directly to instructions in the native architecture, and since in general the register sets of the virtual machine generally match that of the actual hardware, most instructions can execute in one native instruction, as shown in the table. In Table 3, 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, thus it's average execution cost is not applicable in this case.

8. Conclusion and Future Work

In this paper, we present Scylla, a smart yet simple virtual machine architecture for mobile embedded systems, along with a preliminary evaluation of it's performance. The key features of Scylla are communication support, power management and graceful error recovery.

The communication primitives enable tasks to be loaded from task sources and migrate to other devices in an architecturally independent manner. This communication may take place over some shared (wireless or wired) medium, or may be communication with a device attached to a processor's general purpose I/O port. Scylla can estimate the energy cost of a task when the task is loaded and compiled. If the energy is underestimated, Scylla can detect this at runtime and gracefully recover by executing the fault handler that the task must provide. The fault handler can simply clean up and terminate the task or it may migrate the task to another device. The virtual machine guarantees that once a task is accepted for execution, there will always be sufficient system resources to execute it's fault handler, and the fault handler will execute to completion. To enable these features, the simplicity of the virtual machine's instruction set is crucial. Scylla achieves this by closely matching the architecture of the virtual machine to that of 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 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 1KB in size. Energy measurements are performed using an off-the-shelf digital multimeter, and instruction usage statistics obtained from an architectural simulator. An evaluation using a synthetic application showed that the cost of on-the-fly compilation is insignificant compared to the cost of execution, assuming moderate durations, even though the cost of the execution was in itself small. We are actively pursuing ways to improve the performance and energy cost of the system. One direction we are currently investigating, is the ability to proactively put the device into a low power mode. Most modern microprocessors and microcontrollers support one or more such modes, and it would be prudent to take advantage of this facility. Also missing from our current methodology is the ability to accurately estimate the cost of communication. We are currently looking further into performing more detailed communication and transmission protocol energy analyses. Finally, we are also designing a simple operating system and communication protocol that will utilize the architecture described here, to enable interesting applications in distributed mobile and embedded systems.

9. REFERENCES

[1] I. B. M. Corporation. *Virtual Machine/Enterprise Systems Architecture, General Information, Version 2 Release 4.0.* International Business Machines Corporation, May 1999.

[2] 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.

[3] 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.

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

[5] W. R. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptaive 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.

[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] R. Meyer and L. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9(3):199–218, 1970.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] M. Varian. Vm and the vm community: Past, present, and future. In *SHARE 89, Sessions 9059-9061.* SHARE, 1989, Revised August 1997.

[14] W. Webb. Embedded java: an uncertain future. *Electronic Design News*, 44(10):89–96, May 1999.

[15] P. Winterbottom and R. Pike. The design of the inferno virtual machine. In *Hot Chips 9 Symposium*, August 1997.