USENIX Association

Proceedings of the Third Virtual Machine Research and Technology Symposium

San Jose, CA, USA

May 6–7, 2004

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

A Virtual Machine Generator for Heterogeneous Smart Spaces

Doug Palmer

CSIRO ICT Centre

`Doug.Palmer@csiro.au`

Abstract

Heterogenous smart spaces are networks of communicating, embedded resources and general-purpose computers that have a wide spread of power and capabilities. Devices can range from having less than a kilobyte of RAM to many megabytes.

Virtual machine techniques can be used to control some of the inherent complexity of a heterogeneous smart space by providing a common runtime environment. However, a suitably rich, single virtual machine implementation is unlikely to be able to operate in all environments.

By using a virtual machine generator and allowing virtual machines to be subsetted, it is possible to provide numerous virtual machines, each tailored to the capabilities of a class of resources.

1 Introduction

A heterogeneous smart space, such as the SmartLands[18] smart space contains many different sensors and controllers, each with their own set of capabilities and, in particular, computing power. Individual devices can range in power and size from a Berkeley Mote (128Kb flash memory, 4Kb SRAM)[4] to a PDA (64Mb RAM)[16]. The smart space, as a whole, can also have access to general-purpose computing resources[21].

A contrast to a heterogeneous smart space is the sort of homogeneous smart space as the Ageless Aerospace Vehicle skin[14], or a Motes network, where the computing resources available tend towards uniformity.

Heterogeneous smart spaces can be expected to appear whenever longevity and cost are overriding issues; in a farm or building, for example. There are a number of factors driving heterogeneity in these environments:

- Pre-existing resources may be built into the smart spaces environment — sensors in the fabric of a building, for example — and difficult to replace or upgrade. A sensor and its associated processing element may be expected to last for the lifetime of the smart space, leading to a 20- or 30-year gap between the oldest elements and the latest introductions, with an associated disparity in performance.
- Resources are introduced into the smart space for a variety of purposes — a temperature sensor and an automatic feeding gate, for example — and may be selected for reasons other than compatibility.
- Those resources that can be upgraded will, most likely, be upgraded piecemeal, when funds and suitable products are available.
- The purpose of the smart space environment may change. For example, a warehouse may be sold and renovated as residence.

The environments which generate heterogeneous smart spaces also tend to generate a plethora of distinct applications, all competing for resources. On a farm, for example, the stock protection, environment monitoring and irrigation systems may all want to use a single temperature sensor for a variety of purposes. New applications may be added, and old ones removed, in an ad hoc manner. These applications will tend to be of ordinary commercial quality, rather than safety-critical quality and will often fail or go awry; the smart space as a whole will need to be protected from rogue applications.

A feature of heterogeneous smart spaces is that common applications — building heating, stock protection, active maps, etc. — need to be deployed into individual, complex smart spaces. To allow smart spaces to be useful at a common, commercial level, some mechanism for automated customisation and deployment is needed. There are two strands to automated customisation and deployment: at the top level, a declarative service description language model is needed, to allow applications to abstract the resources needed to perform a task[17]; at the bottom level, some sort of mechanism is needed to help control the complexity inherent in an ad hoc collection of resources with competing applications.

Figure 1 shows an example top-level deployment onto a field smart space. The smart space consists of some low-level soil moisture sensors with minimal processing power and range, some intermediate-level fence-post processors and a general-purpose monitoring and management facility. The moisture sensors have been "sown" into the field over several years. Each sowing uses whatever agricultural sensor packages are most economical at the time, leading to a mixture of architectures and processing platforms. The sensors form an ad-hoc network with each sensor connecting to any near neighbours.
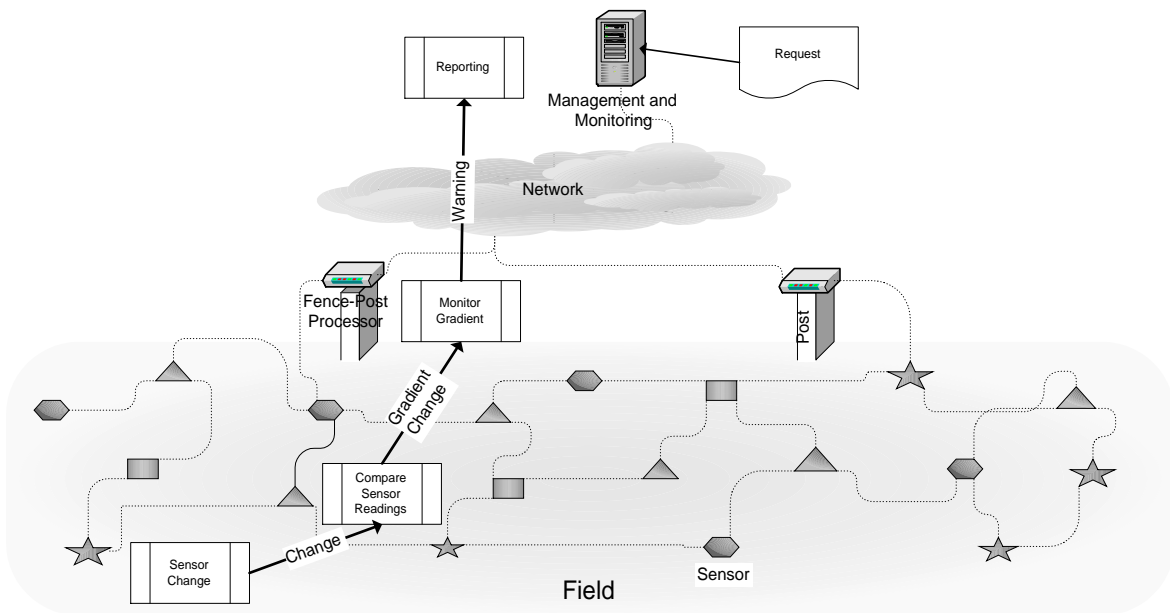
Figure 1: Example Smart Space Deployment

At the top level is a declarative request made by an application to monitor the moisture gradient of the field and raise an alarm if the gradient fluctuates outside acceptable bounds. This request must be mapped onto the smart space by the smart space itself, since the smart space is aware of the resources available, their capabilities and their properties. In the example smart space, sensors report any significant changes in moisture content to neighbours, which then compare the changes with their own readings. If a fluctuation is detected, the event is reported to a nearby fence-post processor, which collates reports in a local area and notifies the monitor of any significant changes. The deployment shown in Figure 1 only shows one instance of each routine for clarity. Each sensor is running both the sensor monitoring and gradient change detection routines.

A consequence of the request is that essentially identical programs need to be run on a wide range of hardware platforms, corresponding to the range of sensors that have been distributed in the field. A bottom-level system that allows a separation between program and implementation would help control the complexity inherent in a deployment across multiple resources. The main requirements for such a bottom-level system can be summarised as follows:

heterogeneity Multiple source representations and multiple machine architectures need to be accommodated. A wide range of computing power and space needs to be supported.

parsimony The system needs to be able to fit into the very limited resources available on some smart spaces environments.

economy Power consumption and network traffic need to be kept to a minimum.

security Hostile or buggy code should have minimal impact.

concurrency Multiple applications may need to run independently on a single resource. A single application may not hog all the resources available.

The advantages of a common language runtime have long been recognised when working with many machine architectures and many languages[15]. A virtual machine allows a *safe* common language runtime to be implemented, with the virtual machine preventing overflows and illegal, unmediated access to resources such as sensors, processor time and memory not allocated to the program being run.

However, some of the computing resources in a smart space are not large enough to handle dynamic strings, let alone something as sophisticated as a full object-oriented environment. There is also a considerable difference in the sophistication required across the range of resources. In the soil moisture monitoring example, there is a considerable difference between the simple monitoring functions performed by a sensor and the more complex array processing required in the fence-post processor, where "significance" is determined.

The approach taken here is to make use of the communications inherent in a smart space. Small resources can use a subset of the full virtual machine, perhaps only capable of simple integer arithmetic. More complex processing can occur on larger resources, capa-
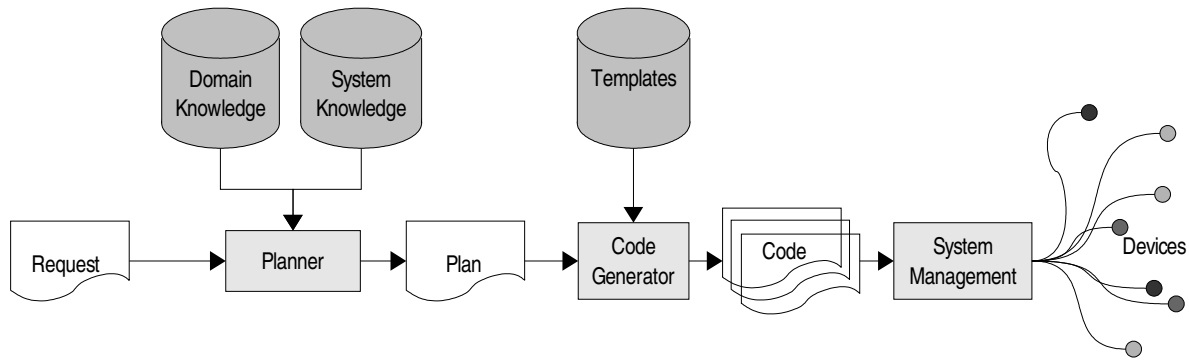
Figure 2: Example Smart Space Code Deployment

ble of more sophisticated processing and memory management. Large scale data and system management can be handled by general-purpose computers[21] or by exploiting the emergent properties of multi-agent systems[14]. An application can be partitioned into fragments of code that can be distributed throughout a smart space, with the low-capability resources offloading sophisticated processing onto their more powerful brethren.

To allow specialised virtual machine subsets, a virtual machine generator is used. An abstract virtual machine specification, along with a description of the subset needed for a particular resource, is fed into the generator. The generator then constructs source code (in C or Java) for a virtual machine that implements the specification. This virtual machine can then be compiled, linked with a resource-specific kernel and loaded into the resource. Application-specific code can be loaded into the running virtual machine across a communications network as components[20].

A sample deployment architecture is shown in Figure 2. Each device has a customised virtual machine, with knowledge about the capabilities of that virtual machine kept in a system knowledge database. A high-level request is given to a planner. The planner uses knowledge about the structure of the smart space and the domain of the request to build a plan: a set of small components (a few subroutines in size) in an intermediate language such as Forth or a subset of C. The plan reflects the known capabilities of the devices and the connections between the devices. Each part of the plan is compiled into code by a templating code generator, which selects code generation templates based on the capabilities of the target virtual machine. The code can then be distributed to the target devices.

1.1 Related Work

Berkeley Motes provide a consistent model for the development of smart spaces. Since Motes have a very small memory footprint, there have been several developments designed to operate in such a constrained environment.

The TinyOS[12] operating system has been developed to provide support for Motes. The nesC[10] language is a language oriented towards TinyOS applications — and TinyOS itself. TinyOS/nesC is designed to support complete static analysis of applications, including atomic operations and data race detection, to ensure reliability. A single application is linked with TinyOS and deployed as a single unit. This approach can be contrasted with the approach taken in this paper, which assumes multiple, dynamic applications and the ability to kill (and reload) misbehaving components.

The obvious advantages of using virtual machines in smart spaces has led to the development of Maté[13] for networks of Motes. There is a considerable overlap between Maté and the virtual machines described in this paper: stack-based, active messages, small footprint. However, Maté follows the general Motes philosophy: a single program and an essentially homogeneous environment allowing a single virtual machine implementation and instruction-level access to hardware.

Virtual machine generation has been used successfully with the VMGen[7] system, used to generate virtual machines for GForth. The virtual machine generator presented here shares many of the characteristics of VMGen, although VMGen performs sophisticated superinstruction generation and does not permit subsetting.

The Denali lightweight virtual machine model[23] offers a similar model to that discussed in this paper: lightweight, multiplexed virtual machines acting as monitors and sharing resources across a network. However, the focus of Denali is on providing isolated multi-
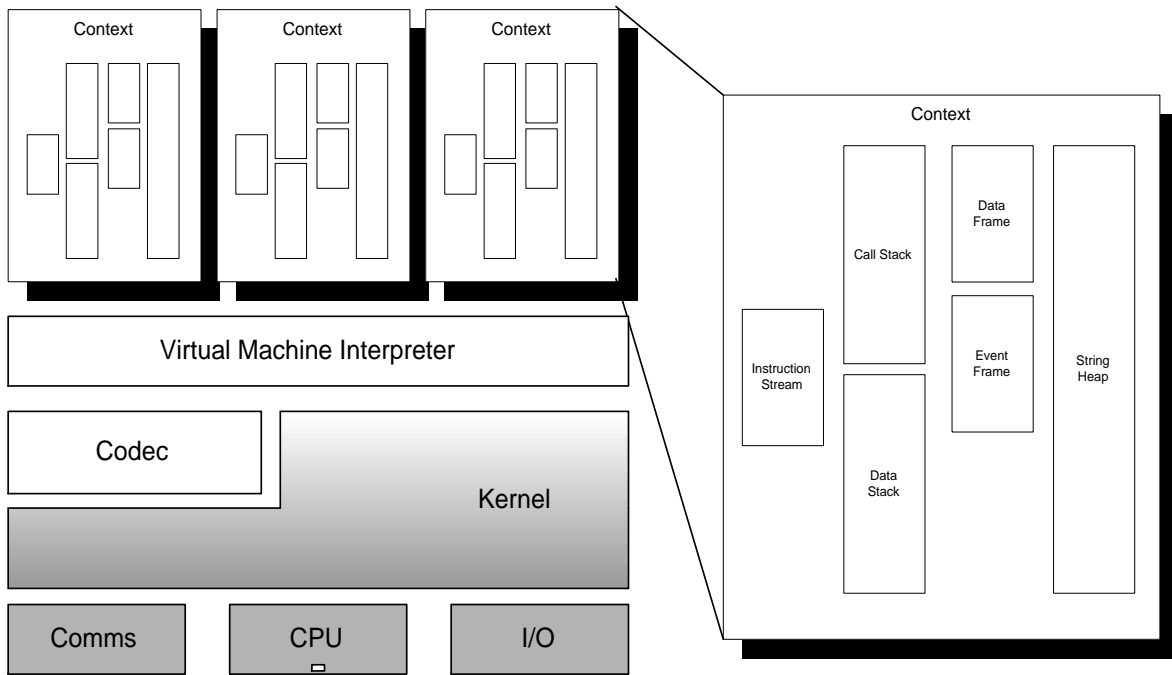
Darker blocks are more hardware-specific.

Figure 3: Generic Virtual Machine Architecture

plexing on large, general-purpose systems and the paravirtualisation techniques used in Denali would compromise the goal of a common runtime.

Also suitable for larger embedded devices is Scylla[19]. Scylla uses a register-based virtual machine that can be easily compiled into the instruction set present in larger embedded processors, such as the ARM. Scylla is oriented towards (just-in-time) compiled applications, something beyond the power of many of the resources discussed in this paper.

1.2 Overview

The paper is structured as follows: Section 2 gives a description of the generic virtual machine architecture that is supported, a stack-based virtual machine with a range of specific data stores; Section 3 describes the way a specific virtual machine is declared in an XML document; Section 4 discusses code generation from a specification to compiled virtual machine, along with some discussion of the size of the generated virtual machine and of potential optimisations; Section 5 concludes the paper.

2 Architecture

The generic virtual machine architecture is shown in Figure 3. The generated virtual machines are stackbased, for the reasons outlined in [6]: ease of code generation and lack of instruction decoding overhead. The generic architecture contains a number of elements: contexts that contain the state of a component; a virtual machine interpreter; a coder-decoder (codec) for marshalling and unmarshalling events; a platform-specific kernel and hardware support (communications, processing and I/O for sensors and actuators). Communications with the outside world, either as direct I/O or across a network link are handled in terms of events. These elements are discussed in more detail below.

2.1 Contexts

Contexts provide a complete state description of a virtual machine component. Since a resource may be managing several components, multiple contexts are supported, with the virtual machine interpreter multi-threading nonpreemptively between them.

In addition to some state variables governing error handling and timing, a context consists of a number of *stores* of various types:

stack A LIFO stack. Stacks that grow upwards or downwards are supported. Two distinguished stacks are the data stack, the default stack for storing operands, and the call stack, used to manage subroutine and event management calls.

stream A stream of data or instructions. Unlike a stack, a stream is assumed to have a single direction, with each read returning the next element of the stream. The distinguished instruction steam is used to provide a stream of instructions for the interpreter.

frame An indexed data frame. Frames contain data in a fixed position that needs to be accessed by a component. A special frame is the dispatch frame, used to store pointers to subroutines that service events.

heap A garbage-collected heap for storing variable-length string or binary data.

Heaps are garbage collected by a conservative mark and sweep, non-compacting algorithm that performs stack and frame walks, looking for possible heap references[3]. Some of the easier misidentification avoidance techniques have been used[2]. Pointers are aligned and references to heap objects are given an unlikely signature, to avoid too many spurious references being identified. Ignoring compaction, while increasing the risk of fragmentation, removes the need for a separate object table.

Stores have an associated data type (eg. 32-bit integer, program instruction pointer) that can be used to translate data moving between the various stores. Stores can be designated as read-only, providing a hint that the store could be placed in flash memory.

2.2 The Loader

Contexts have a loader format that allows a context to be transmitted over the network as a stream of binary data. The loader format lists the various stores that need to be loaded, including the instruction stream.

The loader format is designed to minimise network message size and initialisation code. No relocation is needed, since all addresses are relative to the start of the store. Each store can be installed with the bottom (or top) part of the store pre-initialised; even a stack or heap can be pre-initialised before the program starts. Parts of the store that are not pre-initialised are initialised to a default value, to ensure application isolation. Each store is supplied with an expected size, the size of any pre-initialised data and some information on the expected type and data-type of the store, for basic consistency checking.

Installing a context involves allocating space for the various stores and initialising them from whatever data is supplied to the context. The context can then be added to the scheduling list for the virtual machine interpreter.

2.3 The Interpreter

The virtual machine interpreter is responsible for managing the scheduling of contexts and kernel functions, and the execution of a context's instruction stream.

The interpreter executes in a non-preemptive fashion, with certain instructions causing the interpreter to yield. A specification option also allows yielding after a fixed number of instructions, ensuring good behaviour in an untrusted environment. External events, such as timers, sensor triggers or network messages are handled

by buffering the incoming data until the interpreter is prepared to process it as an event while switching contexts.

Atomic sections of code can be created by preventing yielding. If yielding is not forced after a fixed number of instructions, then atomic sections simply consist of sequences of non-yielding instructions. In this case the code generator for the virtual machine programs needs to be trusted to perform a suitable yielding analysis. If yielding is forced after a fixed number of instructions, it is possible for a yield to occur within an atomic section. A specification option allows a flag to be included in the virtual machine that will cause the program to be immediately rescheduled after yielding. If the flag is not released after a specified time period, then the program is assumed to be malfunctioning and is terminated.

This approach can be contrasted to that of nesC[10], where explicit atomic actions and data race detection is built into the language. The approach taken here pushes the complications of managing hardware-specific functions onto the kernel developer (see Section 2.5) and the issues of ensuring yielding onto the code generator.

2.4 Events and the Codec

Communication with the outside world, either via network messages or through the resource's I/O facilities is handled via events. The virtual machine supports a set of named events, each with an explicit set of parameters.

An outgoing event is sent either across the network or to a service routine in the kernel, where it is applied to the resource. An incoming event is handled by a service routine in a virtual program. The service routine is supplied the event arguments and is expected to capture the arguments and handle any specific responses.

The codec (coder-decoder) is responsible for translating events from/to the stores of a context. To code an outgoing event, the event parameters are retrieved from the data stack and then either marshalled into a message or passed on to the kernel. To decode an incoming event, the incoming message or hardware event is unmarshalled and the event parameters pushed onto the data stack. A call to a service routine, chosen from the dispatch frame, is then inserted into the context and the context is scheduled. When the context is next processed, it will interpret the service routine before returning to the main program thread. To prevent reentrant events, event handling routines need to be atomic (see Section 2.3).

The approach taken is similar to that of active messages[22]. Each message is identified by a type and decoded according to the supplied type. Decoding is done by the codec routines, rather than by the service routines. By decoding the message early, the message buffer can by recycled immediately, rather than needing

```xml
<type ID="int" prefix="i" stack="data-stack" cell="int32"/>


<stack ID="dataStack" name="sp" type="int" default="true" defaultSize="16"/>

<instruction ID="add">
  <description>Add the two top entries on the stack.</description>
  <argument>v1</argument>
  <argument>v2</argument>
  <result>v</result>
  <operation target="java">
    v = v1 + v2;
  </operation>
  <operation target="c">
    v = v1 + v2;
  </operation>
</instruction>


<event ID="reading">
  <description>Notification of a reading from a sensor.</description>
  <argument>sensor</argument>
  <argument>data</argument>
</event>
```

Figure 4: Example Virtual Machine Declarations

to wait for each context to decode the message individually.

Message addressing is via UUIDs[11]. Each context is given a UUID, allowing simple point-to-point messaging, as well as broadcast.

From the point of view of a context, events that cause messages are indistinguishable from direct hardware events. Treating the two uniformly makes translation between a resource with direct access to sensors and other elements and a resource that needs to make use of other resources relatively straightforward.

2.5   The Kernel

The kernel is the interface between the hardware of a resource and the virtual machine. The kernel is responsible for:

- memory management;
- communications and connection management;
- interfaces to directly implemented events;
- direct output to hardware;
- managing input (synchronous and asynchronous) from hardware; and
- low-level timing

The kernel and virtual machine interpreter run under a single thread. Other threads — or interrupt routines — may handle aspects of I/O and communications. These threads are invisible to the interpreter. The kernel is polled by the virtual machine interpreter for any events while switching contexts. If there are no active contexts, the kernel is responsible for waiting for an event or timeout for the interpreter to process.

3   Virtual Machine Specification

A virtual machine is specified in an XML document. The use of XML allows both ease of use and the wide range of XML tools and technologies to be applied to the specification. The specification allows a stack-based virtual machine to be generated. The essential elements of a specification, shown in Figure 4, are:

type declarations Type declarations allow the creation of logical types, such as `int`. Logical types can be associated with particular primitive types, such as `int32` for 32-bit integers and default store locations.

store declarations Store declarations describe the stacks, heaps and other elements that the virtual machine manipulates.

instructions Instruction definitions describe the input and output arguments of the instruction, along with the stores that the arguments come from and go to. Repeated argument names are assumed to refer to the same value. Code implementing the instruction in the target language (Java or C) can also be given.

events Event definitions are similar to instruction definitions, except that no implementing code is supplied. The implementation of the event is either as a direct kernel function or as a message sent to another resource.

```
<subset>
  <description>Exclude strings</description>
  <include type="store">.*</include>
  <include type="instruction">.*</include>
  <include type="event">.*</include>
  <exclude type="store">strings</exclude>
  <exclude type="instruction">dups</exclude>
  <exclude type="instruction">appends</exclude>
  <exclude type="instruction">str</exclude>
  <exclude type="event">message</exclude>
</subset>
```

Figure 5: Example Subset Declaration

In addition to the basic virtual machine definition, a separate XML document contains a subset declaration for the virtual machine. An example subset declaration is shown in Figure 5. The subset declaration lists those instructions, events and stores that are to be implemented. The subset declaration also, in the case of events, defines them to be direct or message events. Subset elements can be defined either by inclusion or exclusion. For conciseness, the inclusions and exclusions use regular expressions to match store, instruction and event names.

4   Virtual Machine Generation

The virtual machine generation process is shown in Figure 6. A virtual machine specification and subset declaration are fed into the generator. The generator then analyses the virtual machine and generates a series of source code files for Java and C that implement the subset virtual machine. The source files are then compiled and linked against a standard library of support functions and classes. An assembler is also generated. Sample declared instructions and generated C code is shown in Figures 7 and 8.

The complete virtual machine is analysed and instruction codes, event codes and stores are allocated before subsetting. By analysing the complete virtual machine, a subset virtual machine is guaranteed to be compatible with any superset implementation.

Code generation makes extensive use of the Visitor pattern[9]. Each virtual machine construct (type, instruction, store, event, etc.) is represented by an object. A language-specific generator is then used to generate appropriate code.

Superinstruction analysis and generation[7] is not performed. The trade-off in a memory-constrained environment between virtual machine size, on one hand, and code size and speed, on the other hand, is difficult to manage. The aim of the generator is to generate multiple virtual machines, all providing a subset of a common runtime.

4.1   Java Code Generation

Java code generation is relatively straightforward. A separate class file for each element of the virtual machine shown in Figure 3 is generated, along with interfaces for common elements, such as instruction codes. Abstract superclasses provide any common functionality that is needed.

The interpreter uses a large switch statement to decode instructions. For each instruction, arguments are gathered from the various stores and placed in temporary variables. Any implementation code that is part of the declaration is then executed. Any results are then returned to the appropriate stores.

The generated virtual machine interpreter moves commonly used context elements (stack pointers, store arrays) to temporary variables while the context is being executed. These variables are replaced whenever the interpreter cycle for that context finishes or when an instruction with side-effects — such as an event send — is executed. The more sophisticated stack caching techniques, discussed in [6], are not implemented, although implementing them would clearly improve performance and caching behaviour.

The UUID method of addressing has proved cumbersome. It is difficult to handle 128-bit objects efficiently without generating large amounts of code, special instructions and special stores. A local context identifier that fits the natural data size of the virtual machine would seem to be more useful, at the expense of more management complexity at higher levels.

4.2   C Code Generation

The C code generator generates code that is very similar to the generated Java code. The main difference between the two generators is that structs, rather than classes, are used for data structures, with functions taking the structs as arguments. Library code is in the form of individual functions, rather than abstract classes. C, rather than C++, is generated, so that a minimalist approach can be taken to object construction and destruction.
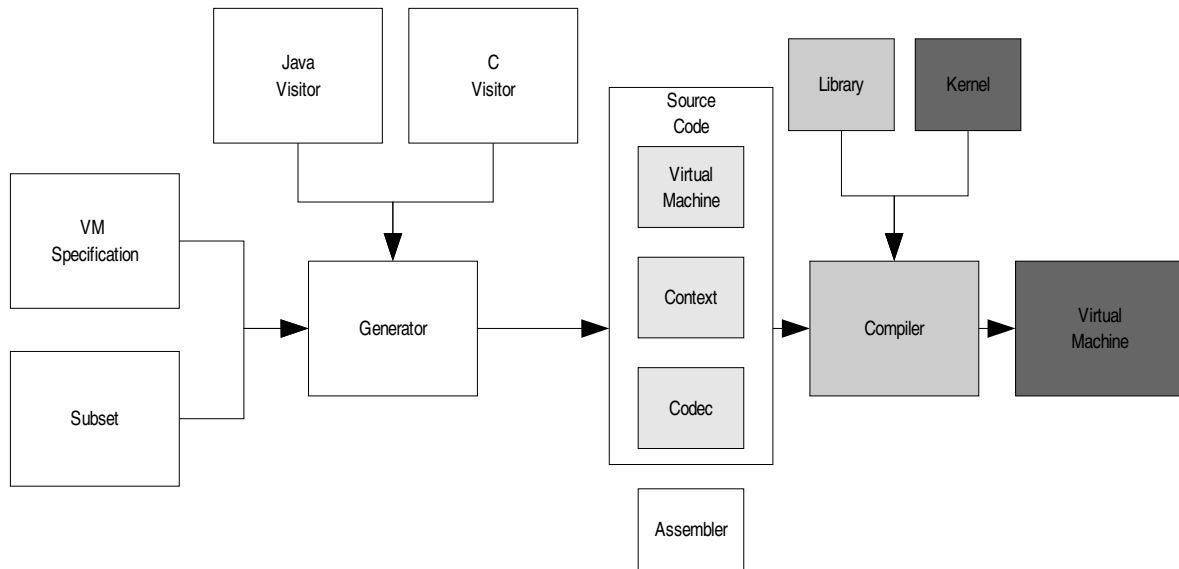
Figure 6: Virtual Machine Generation

```
<instruction ID="dup">
  <argument>v</argument>
  <result>v</result>
  <result>v</result>
</instruction>
```

(a) instruction declaration

```
case I_DUP:
  if (sp + 1 > dataStackSize || sp < 1)
    goto stack_error;
  _temp1 = dataStackData[sp++];
  dataStackData[--sp] = _temp1;
  dataStackData[--sp] = _temp1;
  break;
```

(b) generated code

Figure 7: Code generation for the dup instruction

```
<instruction ID="str">
  <argument>v</argument>
  <result heap="strings">r</result>
  <operation target="java">
    r = Integer.toString(v);
  </operation>
  <operation target="c">
    itoa(scratch_buffer, SCRATCH_BUFFER_SIZE, v);
    r = scratch_buffer;
  </operation>
</instruction>
```

(a) instruction declaration

```
case I_STR:
  if (sp + 1 > dataStackSize)
    goto stack_error;
  _temp1 = dataStackData[sp++];
  itoa(scratch_buffer, SCRATCH_BUFFER_SIZE, _temp1);
  _temp6 = scratch_buffer;
  _temp2 = _temp6 == NULL ? 0 :
    heap_storeString(context->strings, _temp6);
  if (_temp2 < 0)
    goto heap_error;
  dataStackData[--sp] = _temp2;
  break;
```

(b) generated code

Figure 8: Code generation for the str instruction

	Library Code				
Class	Java		C		Description
	Full	No Strings	Full	No Strings	
Basic Heap	0	0	643	643	Core heap management
VM Base	1259	1250	690	690	Common virtual machine functionality
Kernel Base	29	29	80	80	Basic kernel functionality
Codec Base	268	268	916	740	Common coder-decoder functionality
Connection	1335	1158	531	531	Communications management, marshalling and unmarshalling
Heap Manager	1705	0	878	0	Garbage-collected heap management
Loader	1365	1078	955	741	Context unmarshalling and loading
UUID	539	539	68	68	UUID implementation
	6500	4331	4761	3493	
	Generated Code				
Class	Java		C		Description
	Full	No Strings	Full	No Strings	
Codec	1252	1134	1215	839	Generated coder-decoder
Context	786	530	787	525	Generated context
Kernel	660	660	801	801	Kernel for 3 LEDs, a temperature sensor and a heat pump
VirtualMachine	1777	1527	1913	1430	Generated virtual machine
	4475	3851	4716	3595	

Table 1: Code Sizes for a Generated Virtual Machine

The C virtual machine interpreter needs to do a great deal more bounds checking than the Java interpreter. Stacks, for example, may not overrun their boundaries — something guaranteed by the Java virtual machine.

4.3 Code Size

The code generated is relatively compact. Table 1 shows the relative code sizes for a simple virtual machine with and without string handling. The Java code was generated by the Sun 1.4.2.01 javac compiler. The C code was generated for a Pentium 4 processor by gcc 3.3.2 with the -Os option. Total size is 9–11k bytes of code for the virtual machine with string handling and 7–8k for the same machine without string handling.

The full virtual machine contains 29 instructions, 6 events, a data stack, a call stack, a data frame, a dispatch frame, a string heap and an instruction stream. The stringless virtual machine contains 25 instructions, 6 events, a data stack, a call stack, a data frame, a dispatch frame and an instruction stream. The underlying resource is a simple resource with 3 3-colour LEDs, a temperature sensor and a heat pump.

String handling increases the size of the generated virtual machine considerably. Clearly, a heap manager is needed, which increases code size. However, string management tends to be more complex in general, requiring specialised marshalling and unmarshalling and more complex instruction implementations. The method size in both the Connection and Codec classes increases by approximately 50% whenever string handling is needed. More importantly, given the small amount of RAM available, string handling requires the allocation of blocks of memory to act as a heap.

The network management and message passing parts of the virtual machine take up a significant part of the total memory footprint. Message and program transmission can be considered a relatively rare event — or, at least, it should be, if energy consumption is to be taken into account — in which case its influence on caching and power consumption (see Section 5) can be regarded as negligible. However, it would be a good thing, on principle, to reduce the amount of code needed for such an operation. At present, marshalling is handled by dedicated routines, one to each type of message. An alternative is to try an data-driven, interpreter-based approach[5]. If there a large number of events, this approach looks attractive.

An assembled program takes up little space. Table 2 summarises the context sizes, in the network deliverable loader format (see Section 2.2), for a number of simple programs.

The sizes shown in Table 2 show the minimum amount of information needed to initialise a context. Installed contexts usually take up more space within the resource: stacks need enough room to grow and heaps usually need additional space for new blocks of data.

Program	Size (bytes)	Description
ChangeReport	134	Polling report of sensor change
EventReport	116	Event-driven report of sensor change
Chaser	166	LED chaser
AirCon	167	Simple airconditioning

Table 2: Assembled Application Code Sizes

5 Conclusions and Further Work

The diversity and complexity of heterogeneous smart spaces, coupled to the stringent restrictions on resource usage that networks of small embedded devices imply, presents a considerable software engineering challenge. The sort of component reuse strategies that have become common in commercial programming environments will also need to be applied to smart spaces, if smart spaces are to become general-purpose, commercial environments. The use of virtual machines provides a method for distributing generic functionality across a wide range of resources.

There are a number of virtual machine optimisations and improvements that could be undertaken. These optimisations are discussed in Sections 4.1 and 4.3. In particular, code-size optimisations can be expected to play an important part in reducing the size of the generated virtual machine. An advantage to using a generator is that any optimisations that are made will propagate to any newly generated virtual machine, rather than requiring hand-optimisation.

Energy consumption and power management is a major concern in the space of small embedded devices, with memory access a significant source of energy consumption. Testing of the energy consumption of Java virtual machines in the Itsy pocket computer suggests that there is the order of a 50% penalty in energy consumption when interpretation is used, instead of a just-in-time compiler[8]. There is an order of magnitude difference between cache memory access and external memory access, however[1]. If the virtual machine interpreter — or a subset of frequently used instruction implementations — and a context could be fitted into cache memory, the energy costs could be significantly reduced. The compression effect of virtual machine instructions would then serve a useful purpose in allowing a component to be entirely cached.

Generating virtual machine subsets allows a common runtime environment to be imposed on the diverse array of resources that make up a heterogeneous smart space. Using a generator allows virtual machines to be quickly generated for new resources and to try new instruction sets. The generated virtual machine is relatively compact, although there is considerable room for improvement.

References

[1] Luca Benini, Alberto Macii, and Massimo Poncino. Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. *ACM Transactions on Embedded Computing Systems*, 2(1):5–32, February 2003.

[2] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 197–206, Albuquerque, New Mexico, June 1993.

[3] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9):807–820, September 1988.

[4] Crossbow: Wireless sensor networks, 2003. http://www.xbow.com/Products/Wireless_Sensor_Networks.htm.

[5] K.V. Dyshlevoi, V.E. Kamensky, and L.B. Solovskaya. Marshalling in distributed systems: Two approaches, June 1997. http://www.ispras.ru/~microrb/papers/index.html.

[6] M. Anton Ertl. Stack caching for interpreters. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, La Jolla, California, June 1995.

[7] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen — a generator of efficient virtual machine interpreters. *Software — Practice and Experience*, 32(3):265–294, 2002.

[8] Keith I. Farkas, Jason Flinn, Godmar Back, Dirk Grunwald, and Jennifer M. Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 252–263, Santa Clara, California, June 2000.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, California, June 2003.

[11] The Open Group. *DCE 1.1: Remote Procedure Call*, October 1997. Standard C706, http://www.opengroup.org/onlinepubs/009629399/toc.pdf.

[12] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for network sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Cambridge, Massachusetts, November 2000.

[13] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 85–95, San Jose, California, October 2002.

[14] Howard Lovatt, Geoff Poulton, Don Price, Mikhail Prokopenko, Philip Valencia, and Peter Wang. Self-organising impact boundaries in ageless aerospace vehicles. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2003)*, pages 249–256, June 2003.

[15] Stavros Macrakis. From UNCOL to ANDF: Progress in standard intermediate languages. Technical report, Open Software Foundation, 1993.

[16] Palm, inc., 2003. http://www.palm.com.

[17] Doug Palmer. Declarative application programming in smart spaces. Technical Report 03/88, CSIRO Mathematical and Information Sciences, January 2003.

[18] Smartlands, 2003. http://www.smartspaces.csiro.au/applic/smart-lands.htm.

[19] Phillip Stanley-Marbell and Liviu Iftode. Scylla: A smart virtual machine for mobile embedded systems. In *Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'00)*, pages 41–50, Monterey, California, December 2000.

[20] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[21] Ken Taylor and Doug Palmer. Applying enterprise architectures and technology to the embedded devices domain. In *Proceedings of the Workshop on Wearable, Invisible, Context-Aware, Ambient, Pervasive and Ubiquitous Computing (WICAPUC)*, number 21 in Conferences in Research and Practice in Information Technology, Adelaide, Australia, February 2003.

[22] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. Technical Report USB/CSD 92/#675, University of California, Berkeley, March 1992.

[23] Andrew Whitaker, Marianne Shaw, and John D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical Report 02-02-01, University of Washington, 2002.