Virtual Machines for Ambient Computing: A Palpable Computing Perspective

OT4AmI ECOOP Workshop Position Paper

Ulrik Pagh Schultz Erik Corry Kasper V. Lund
PalCom, http://www.ist-palcom.org
University of Aarhus Esmertec
Denmark

June 3, 2005

Abstract

Ambient computing promises to deliver a smooth end-user experience where computers integrated into the environment automatically and transparently support users in their daily tasks. Users must however always remain in control; balancing transparency and automation with awareness and control is the goal of the "Palpable Computing" project (http://www.ist-palcom.org). As an extension of ambient computing, PalCom places requirements on many parts of the system architecture; we here address the requirements placed on the underlying virtual machine. Specifically, we describe the overall design and features of our current prototype implementation of a virtual machine for palpable computing systems, and outline our approach to resolving critical issues within reflection, scalability, and stability.

1 Introduction

Ambient computing promises to deliver a smooth end-user experience where computers integrated into the environment automatically and transparently support users in their daily tasks. We believe however that users must always remain in control; balancing transparency and automation with awareness and control is the goal of the "Palpable Computing" project (PalCom, http://www.ist-palcom.org, details later in the paper). Palpable computing complements key features of ambient computing systems, such as invisibility and end-user composition of devices, with dual features (e.g., visibility and decomposition) that enable users to navigate and influence the computing system.

PalCom (and ambient computing in general) places requirements on many parts of the system architecture, including the division of a computing system into individual *services*, the distributed communication protocols, the underlying component model, etc. It is however not obvious to what extent PalCom places requirements on an underlying virtual machine (a virtual machine can optionally be used on PalCom devices to provide the usual advantages in terms of portability and mobility of compiled code.) Nevertheless, we believe that directly supporting certain core qualities of PalCom in the virtual machine will vastly facilitate the implementation of the higher-level layers of the system.

We are currently developing a prototype virtual machine for palpable computing systems, as part of the PalCom project. In this paper we describe the overall design and features of our implementation, and outline an approach to resolving two critical issues in palpable computing: reflection and implementation decoupling (the latter is essential for achieving scalability, and stability). We believe that our approach is general in the sense that it is applicable to ambient computing in general and that the same techniques can be generally useful in virtual machines.

Note. The virtual machine described in this paper is also one of the topics of the paper "An Open Architecture for Palpable Computing: Some Thoughts on Object Technology, Palpable Computing, and Architectures for Ambient Computing" also submitted to this workshop.

Context: PalCom

Palpable computing can be seen as extending ambient computing with additional characteristics for user control. We are in the context of this paper only interested in a few of these characteristics; for a more complete treatment we refer to the PalCom website [10]. Palpable computing systems offer not only *invisibility* (the capacity of unobtrusively performing computing tasks in the background environment) but also *visibility*, that is, the capacity of making visible to users what they are doing and what they may do. Moreover, systems should offer both *construction* (the ability to support end-user composition of devices or services to form new devices and/or services) and also *deconstruction*, that is, the ability to disassemble a device or service into its constituent parts to enable understanding and manipulating each part individually.

2 PreVM

We now describe our prototype virtual machine for palpable computing systems, dubbed "PreVM" (Palpable runtime environment Virtual Machine). The goal of PreVM is to match the evolving needs within the PalCom project, resulting in a specification and reference implementation of a virtual machine for PalCom devices.

PreVM is a language-neutral virtual machine designed to support object-oriented languages. PreVM is dynamically typed, which requires each language to implement its proper type checks (e.g., the "instance of" relation in Java), but on the other hand imposes no restrictions on the type system of the languages that it supports. For example, languages such as Beta and Java 1.5 that sport type systems with different forms of covariance can both be implemented on the same virtual machine without incurring redundant type checks. The underlying virtual machine does however enforce the same safety rules found in Smalltalk: integers cannot be manipulated as pointers, only existing fields can be accessed from an object, and only methods declared in the class of an object can be called on the object.

PreVM programs are deployed in binary components which are instantiated as *run-time components*, objects that encapsulate a set of classes and their required and provided interfaces. We have implemented source-language-to-binary component compilers for the Smalltalk, Java and Beta languages [6, 7, 8]. Interoperability between programs written in each language is currently restricted, but we are working on an approach based on interface specifications to permit high-level and type-safe interaction between components written in different languages. Note that due to restrictions on the physical characteristics of devices (most notably the available memory), standard libraries from each of these languages cannot be used. Rather, we are developing a common set of fine-grained components encapsulating common library functionality required by PalCom devices.

PreVM is implemented in Java, but is currently being reimplemented in C++. Core PalCom functionality such as communication and discovery is implemented as components running on the virtual machine. Based on previous experience with the OSVM embedded Smalltalk product from Esmertec, the target embedded device is a 32-bit processor with 128K of physical memory [1, 3].

3 Reflection

Features such as visibility, construction, and deconstruction require the underlying system to exhibit a significant degree of flexibility; this flexibility can be implemented using reflection. Reflection for object-oriented languages ranges from the more restricted (and efficient) style of Java to more unrestricted (and inefficient) meta-object protocols; in both cases reflection is performed through an object-centric interface. As an alternative, we propose the use of a data-centric approach, where all data that may be relevant for reflection are stored in a repository, the hierarchical map. A *hierarchical map* (or h-map for short) maps a hierarchically organized namespace to values.[1] The h-map can be accessed both locally and remotely, although a more restricted interface based on asynchronous messaging is used for remote access.

For example, the virtual machine stores all information regarding components, their run-time composition and the classes they contain, in the h-map. This use of h-maps is illustrated in Figure 1. The virtual machine uses this information directly during execution of the program. Reading values from the

[1]This use of h-maps is inspired by the Corundum PalCom prototype [13], which again is inspired by the Plan-9 operating system [11].

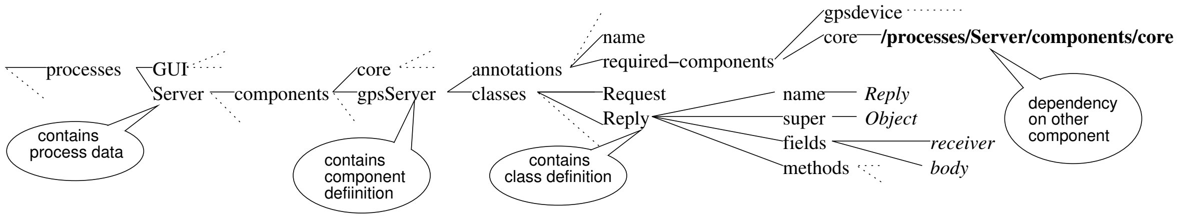

Figure 1: Components and classes stored in the h-map

h-map enables an inspector tool to determine the runtime wiring between components. Writing values to the h-map enables dynamically changing the wiring between components. Moving subtrees between component definitions enables deconstructing components and combining them in a new way. Any service running on the virtual machine can similarly expose information to enable reflective behavior.

This proposed use of h-maps will in effect turn the virtual machine and its services "inside out", allowing their configuration to be inspected and modified in arbitrary ways. Architecturally, the h-map is convenient for representing many of the ad-hoc data structures needed internally in the virtual machine. A higher-level reflection interface can be implemented as a separate component that can run locally or remotely on a "reflection server." As a general principle, we postulate that the more a system is built around static checking, the harder it is to dynamically modify and update the system (as an example, consider dynamically modifying the declaration of a class in Java, which is only possible in very specific circumstances). We belive that a system built around a dynamically evolving h-map structure supports unanticipated use better than a more traditional, statically configured system.

It is however an open question how much of the virtual machine state it is useful to store in the h-map; in principle the individual bytecodes and the program counter could be stored, but that is probably not useful in practice. Alternatively, parts of the h-map could be virtual, that is, computed (but never stored) when accessed; this approach would for example allow debugging information (including the program counter) to be accessed through the h-map. Last, as a means to improve efficiency, we believe that program specialization techniques could be employed to specialize the virtual machine implementation to fixed subtrees in the h-map (this approach would primarily be useful for statically configured devices).

4 Implementation decoupling

Palpable computing systems, and indeed any ambient computing system, involve a heterogeneous mix of devices with different capabilities, ranging from standard workstations to tiny, embedded systems. The palpable computing system must provide *scalability* across different devices and *stability* so that errors in one part the system do not propagate to other parts of the system. These features are supported by an appropriate degree of decoupling between different parts of the implementation.

PreVM provides an execution platform for components. Scalability is achieved by appropriate layering of this platform, allowing the functionality needed in a concrete scenario to be dynamically installed into the virtual machine. The platform is divided into two layers, the native layer and the virtual layer. The *native layer* basically provides an interpreter (or JIT compiler) and a memory manager; it is normally implemented in a low-level programming language such as C or C++. The virtual layer runs on top of the native layer, is built from dynamically loaded components, and can thus be written in any language supported by the platform. For example, we have implemented a component loader in Smalltalk, a thread scheduler in Beta (the native layer provides a simple coroutine mechanism [9]), and we are currently implementing a communication stack in Java. In general, since we want to support bare-bones execution without an underlying operating system, OS-level functionality such as device drivers and file systems should also be implemented in the virtual layer. This approach has already been experimentally verified for embedded Smalltalk in the Esmertec OSVM product [1].

We believe that application isolation is one of the cornerstones in providing stability in palpable (ambient) computing systems. Application isolation is however not easy to achieve on small embedded systems where the virtual machine and all core functionality must be shared due to memory constraints. Moreover, such systems normally do not provide hardware support for memory protection, which is typically used for providing isolated address spaces and general memory management in operating systems. Rather, the virtual machine must provide similar abstraction by limiting interaction between applications and the underlying layers [4].

PreVM currently provides support for multiple, isolated processes. Process isolation is however implemented manually, that is, by implementing the virtual machine such that references are not passed between processes and that each process has a copy of the virtual layer. This is currently possible because core functionality such as scheduling between processes is done by the native layer; as more functionality is moved to the virtual layer, it is more likely that programmer errors can introduce direct references between processes or interference between a process and the virtual layer.

We observe that this interference problem is similar to the more general object aliasing problem [12, 2, 5]. Nevertheless, we wish to allow for a process to exchange objects with the virtual layer, but only so long as we can guarantee that the process cannot inappropriately modify structures internal to the virtual layer. One way to solve this problem would be to disallow exchanging anything but immutable data between processes and the virtual layer, but one can imagine other less restrictive solutions to this problem. In general, we are interested in a concept of lightweight processes, where objects can be contained within specific domains; such a concept could also be generally useful internally to processes, to solve some classes of object aliasing problems.

References

[1] Esmertec AG. OSVM. `http://www.esmertec.com/solutions/M2M/OSVM/`.

[2] Jonathan Aldrich and Craig Chambers. Ownershop domains: Separating aliasing policy from mechanism. In *ECOOP'04 Conference Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25, 2004.

[3] Jakob R. Andersen, Lars Bak, Steffen Grarup, Kasper V. Lund, Toke Eskildsen, Klaus M. Hansen, and Mads Torgersen. Design, Implementation, and Evaluation of the Resilient Smalltalk Embedded Platform. *Computer Languages, Systems & Structures*, 31(3-4):127–141, 2005.

[4] G. V. Back and W. C. Hsieh. Drawing the red line in Java. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 116–121, Rio Rico, AZ, March 1999. IEEE Computer Society.

[5] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA'02 Conference Proceedings*, pages 292–310, 2002.

[6] A.J. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison-Wesley, second edition, 2000.

[8] O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented programming in the Beta programming language.* Addison-Wesley, Reading, MA, USA, 1993.

[9] K. Nygaard and O.J. Dahl. Simula 67. In R.W. Wexelblat, editor, *History of Programming Languages*. ACM Press, 1986.

[10] PalCom. PalCom web site. `http://www.ist-palcom.org`.

[11] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Tricky, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[12] Nathanael Schärli, Andrew P. Black, and Stéphane Ducasse. Object-oriented encapsulation for dynamically typed languages. In *OOPSLA'04 Conference Proceedings*, pages 130–149, October 2004.

[13] Peter Ørbæk. Programming with hierarchical maps. Technical Report DAIMI PB-575, DAIMI, 2005.