Challenging VMs on Battery-Powered Embedded Devices

Roel Wuyts [*]

IMEC, Leuven

wuytsr@imec.be

Abstract

Many consumer devices, such as portable game consoles or cellphones, can be described as battery-powered wireless embedded devices. Many of these are not taking advantage of virtual machines, certainly not for their core tasks, instead relying on C or close derivatives for implementing their behaviour. However, faced with software that becomes more and more dynamic and hardware that is increasingly heterogeneous, the move to virtual machines is as necessary as it is unavoidable. From our own experience in developing software for battery-powered embedded devices we describe features that these upcoming virtual machines should possess in order to win over the embedded crowd and be a viable competitor against C, not just the only viable option.

1. Opening Remark

This paper will not advance the state-of-the-art in virtual machine research. Instead it sketches the domain of embedded systems development where virtual machines are sorely needed but do not seem to be catching on yet. In this paper we enumerate what we think are some reasons why this is the case, without handing any solutions at the time. We think this can be of interest to the VMIL workshop, given the fact that virtual machine research is looking into embedded devices (see for example the invited talk on the Maxine virtual machine). We feel, however, that these efforts are not yet ready to replace the entire embedded device development. Our position statement is that embedded software development, especially for battery-powered mobile devices, will embrace virtual machines when these virtual machines are modularized, allow developers to take advantage of hardware features, and offer introspective and optimization possibilities.

2. Introduction

We are moving towards a world where very heterogeneous devices are connected — ranging from miniature to big, from mobile to tethered: servers, mobile terminals, wireless sensors, wearable and implanted devices. Todays systems support ever more demanding software applications, and require the addition of more and more heterogeneous hardware resources to continue to meet these demands. The recent trend of GPU accelerated software in the desktop and server world illustrate this trend. In battery-powered embedded devices it has been a trend for longer, and the platforms are much more diversified and 'exotic', with many being built just for one device. When needed, chips are customized in order to meet performance, power consumption or space requirements.

One of the net results of this is that many chips are only programmable in assembly language. It is for a reason that developers of these devices call C a high-level language, and feel spoiled when a C compiler for their system exists. There certainly is room for virtual machines (see (Neuvo 2004) that discusses the roles for a Java virtual machine on Nokia cellphones), for the same reasons that virtual machines are nowadays underpinning most industrially-relevant programming languages used in the desktop and server space. The catch is that these virtual machines need to make sure that the software they run adheres to various non-functional requirements such as energy consumption, quality of service, and performance.

Constructing virtual machines that allow this is a huge challenge, encompassing engineering issues (execute applications efficiently and not consume too much power), software engineering issues (the virtual machine should be pluggable in order to be customizable with respect to the hardware), programming language issues (what language mechanisms are needed that are amenable for optimized execution on different platforms, controllable by the developer and still high-level), compiler issues, runtime issues, etc.

IMEC has done research in software development for battery-powered wireless embedded devices for quite some time. From this experience the rest of this paper describes some features that we think are an absolute necessity in order for virtual machines to be a viable alternative to low-level compiled languages to program battery-powered embedded devices.

We have bundled the desired features in the following groups:

- Hardware connection: links between the virtual machine and the hardware.
- Modularization: modular virtual machines are needed to adapt them to various hardware platforms or reconfigure them at runtime.
- Introspection and optimization: multi-criteria optimization (e.g. trade of performance vs. energy consumption) possibilities.
- Adaptation: runtime adaptation according to the execution context.

Before we go into detail the following section gives some details about the development issues a current developer faces when developing software for a battery-powered embedded device.

[*] Affiliated with Katholieke Universiteit Leuven, Leuven, Belgium

VMIL'09, October 25, 2009, Orlando, FL.
Copyright © 2009 ACM 978-1-60558-874-2…$5.00

3. Battery-Powered Embedded Device Characteristics

Battery-powered embedded devices share a number of hardware and software characteristics, whether they are camera's, PDA's, cellphones or portable game consoles.

power and energy The dominant hardware characteristic are *power and energy constraints*, that are tied to the fact that these devices are battery powered(van Berkel 2009). As a result, everything, hardware and software alike, is done in order to reduce energy consumption. The obvious goal is to make sure the device can be used for a long time in between charging. However energy consumption is also linked to heat dissipation. Since many of these devices are handheld devices, they simply cannot become too hot to touch. An energy dissipation smaller than 3W is required. Other devices, like audio and video equipment, tries to be passively cooled as much as possible. Controlling energy consumption is key for controlling the amount of heat generated.

area Another important characteristic is *physical space* (area). Consider a high-end cellphone. It should be able to record high-definition video, integrate various wireless network standards, have the possibility to play back all kinds of media formats, play 3D games, and incorporate motion sensors and a GPS. Of course it also comes with many software applications, to make phone calls, instant message friends, integrate with the calendar and e-mail systems at work, etc. Ideally, to consume the least amount of energy possible, dedicated chips for many of these functionalities could be integrated, such as a dedicated Bluetooth chip, 3D gaming chip, sound chip, etc. However all of these chips require physical space, which is simply not possible unless people suddenly feel inclined to run around with a cellphone resembling the first-generation models. The result is that a choice has to be made to use less-specialized but programmable chips for some of these functionalities, even though they are generally less optimized.

heterogeneous platforms Another characteristic is the fact that many different types of processing elements are typically needed (*heterogeneous platforms*) to achieve the required combination of performance and power consumption characteristics (Neuvo 2004). It is very common to develop code that not only uses a CPU, but takes advantage of a dedicated video accelerator chip, as well as a DSP for the audio. Take into account the fact that all of these chips typically have various power optimization features that need to be controlled (so-called *hardware knobs*), and the complexity of even getting an application to run becomes clear.

Partly resulting from these hardware characteristics there are also a number of software challenges.

real-time constraints A first software characteristic that many people think about when they hear embedded software are *real-time constraints*. Indeed, playing video on a device or adhering to wireless networking standards requires timely execution of tasks. Missed deadlines have an impact on the quality of experience of the user (bad network connections or hickups when watching video), or can have life-threatening effects, for example in healthcare applications.

optimization A second software characteristic is *optimization*, which actually goes hand in hand with real-time constraints. While users require advanced functionality of devices, on par with their home PC, the raw computational power and memory are simply not available on these devices. As a result clever coding and optimizations are necessary. Embedded software developers are typically experts in profiling, as well as in deeply understanding the hardware they are using in order to squeeze the last cycle out of their processing element or efficiently use a particular memory element. While a software developer might think in terms of objects placed in memory, an embedded developer will think of a number of bytes placed in a certain part of the memory hierarchy, and will consider the other parts of this hierarchy (registers, level 1 cache, level 2 cache, SD ram, etc.), their connections (32-bit wide bus with particular bandwidths, for example, or a network-on-chip (Dally and Towles 2001) that needs to be configured) and energy consumption (the cost for moving a number of bytes from SD ram to the level 2 cache). Likewise an embedded device developer will need to take the voltage of the CPU into account when developing.

Embedded software development therefore becomes plagued with making trade-offs such as: will we run this piece of code on the DSP, running it at 200 Mhz, or will we run it on the CPU at full speed. The goal is to find the best assignment and scheduling which minimizes the energy consumed to execute the given set of tasks while satisfying timing, precedence and resource constraints of all (sub-)tasks. In the end this boils down to multi-criteria optimization, primarily taking into account the execution times (in order to be able to comply with the real-time constraints) and energy consumption (to not deplete the battery too quickly).

The process of deciding which functionality will run where and with what characteristics is called *task mapping*. Developers typically make several possible task mappings in order to find the best one. However, this mapping process takes a lot of time because each time optimized software needs to be built. Typically this means that a software decomposition is needed such that the resulting parts can be run on different processing elements in parallel. Consider for example a video decoder. It can either run completely on a single core of a CPU, or it can be threaded to take advantage of two cores, or it can be split in even more parts that are run on several cores and accelerators. All of these implementations typically require different data structures and implementations. Trying out a different mapping therefore boils down to reimplementing significant portions of the application. Different mappings are then compared by running and profiling them with test data. This means that the mappings are optimized to process particular runtime scenario's that the developers predicts are likely to occur.

Developers do not have a lot of support to help them with the task mapping problem, apart from experience and profiling tools. There is ongoing research to help with this problem, such as Task Concurrency Management (TCM). TCM (Yang et al. 2001; Wong et al. 2001) is a methodology which balances the resource utilization, performance, and resource manager complexity with a two phase approach (design-time and run-time). TCM helps in finding a good decomposition and balances the resource utilization, performance, and resource manager complexity. To achieve this it uses a two phase approach (design-time and run-time) to select the right granularity of task model in each phase and do the resource management. We are currently in the process to address a main problem in TCM, which is that it does not take shared interconnect resources into account (the bus or network-on-chip (Dally and Towles 2001)) that are used to transfer data between processing elements.

4. VM Features Most Wanted

Given the shortening time to markets and the continuously increasing software complexity, developers need to make trade-offs between the quality of the solution they can deliver and the time and cost for making this solution. It goes without saying that developing everything in assembly language and C for the complicated development issues raised above, is not a very productive process and requires very skilled and experienced people.

Virtual machines can be an answer to these problems, in the same way that they are being used in mainstream development.

However, just porting a virtual machine to a smaller device by limiting its memory footprint will not solve the problems we showed in the previous section. Embedded developers are then simply unable to take advantage of virtual machines and will need to continue to develop in C and assembly instead.

In this section we describe the low-level features we feel that virtual machines have to embrace in order to capture the minds and hearts of embedded software developers. We have not dealt with all issues, such as looking into the programming language aspects, but have focussed on features that have to do with mapping and optimization.

4.1 Hardware Connection

Processing Elements The virtual machine should make it possible to use several hardware processing elements. Most existing virtual machines do not allow this, not even for homegeneous processing elements like multicore CPUs. It should be possible to execute one program on the VM that is then executed on one or more different processing elements.

A number of research projects are looking into virtual machines that take advantage of manycore hardware. All the research we found experiment with the Tilera64 processor, a manycore chip with 64 cores. The *Renaissance* project by David Ungar and Sam Adams proposes a virtual machine able to use 56 of the 64 Tilera cores[1]. We also found work of Stefan Marr on adding concurrency support in instruction sets of virtual machines(Marr et al. 2009). There is also rumors of an Erlang port on the Tilera64[2]. Intel's Tera-Scale environment, that is talking about tens of thousands of cores, is also going to require virtualization and software that can take advantage of multiple cores[3].

While the Tilera64 chip (let alone the Tera-scale project) is not directly usable in portable embedded devices, advances with these homogeneous systems will eventually trickle down to the embedded processor market, where current offerings are already multicore. The ARM11 MPCore, for example, can be configured to have between 1 and 4 cores[4].

This still leaves our initial request open, however, which talked about heterogeneous processing elements, such as CPU's, GPU's, and accelerators. For example, in the world of cellphones the maximum power dissipation combined with a demand for ever more demanding tasks imply that heterogeneous platforms have to be used (Neuvo 2004; van Berkel 2009). This can be seen in the release of ever more capable multi-processor system-on-chip solutions, such as NVidia's Tegra platform[5], that sports an ARM11 CPU, a Geforce GPU, an Image Processor, a HD Video Processor, as well as the memory controller and other infrastructure. Ideally, virtual machines should be able to virtualize the major capabilities of such a heterogeneous setup, which would also benefit desktop and server systems where there is much talk about GPU-accelerated software.

Note that OpenCL is going in that direction, but not far enough. OpenCL makes it possible to write a kernel that can be executed on CPU, GPU and other types of processing elements such as DSPs. This is enabled through virtual machines, where for example NVidia provides a VM for their GPUs as part of their toolkit, similarly to ATI/AMD and Apple. Initially released for desktop machines, for example in Apple's latest *Snow Leopard* version of the OS-X operating system, there are indications that at least one major vendor of mobile graphic chips is working on OpenCL support[6]. While OpenCL makes it possible to execute the same kernel on multiple processing elements (the "write once run everywhere" mantra we all dearly love), the kernel is different from the host language that sets up the necessary memory, decides where to run the kernel, launch it and read back the results. The VMs for the processing elements are also different. Better would be to have a single integrated VM encapsulating all of these processing elements and managing them.

Adaptive runtime resource management technology we are working on could be the basis for this. It dynamically assigns software tasks to heterogeneous processing elements and adapts itself to various runtime situations, maximizing the usage of the processing elements. We have implemented an adaptive runtime resource manager for a server platform consisting of CPU's and GPU's. Initial experiments show that it is able to improve the average execution performance of a multimedia application by up to 30 percent compared with non-runtime managed solutions. We feel this this could be a good candidate to use in a virtual machine for future embedded platforms.

Memory Besides managing processing elements, there should be control over the memory hierarchy, to give a developer fine grained control on what memory is used. This is a difficult subject that has many programming language implications, but is certainly needed.

The reason we think it is needed is that in the past we have managed to reduce power consumption of applications by carefully optimizing the way the memory hierarchy is used. We even have developed a toolset and a methodology in order to let developers optimize their embedded applications, parallelize them, or do both (Mignolet and Wuyts 2009). Key to the approach is that it analyzes the source code, does time and energy profiling, and uses a model to explore different ways of managing the data, for example deciding to copy part of a larger datastructure that resides in one part of the memory hierarchy (say, level 2 cache) to another part of the memory hierarchy (say, a level 1 cache), and process it there. The code is then rewritten so that the appropriate data transfers are done and the loops work on the right data.

Recently this tool was used to manage scratch-pad memories (fast memories close to a processing element and under full control of the developer). A case study applying this tool on an MPEG-4 video encoder showed an overall power reduction of 25%, a 40% power reduction in just the memories and a 40% reduction in processor cycles as compared to an optimized hardware-cache based solution (Baert et al. 2008).

This technology hinges on one very important assumption: it is assumed that the application (or set of applications) that are being optimized are the only ones that are running. While this assumption is still valid for a number of embedded applications and can be slightly relaxed (for example by guaranteeing that part of the complete hardware resources are available to that optimized application), it is no longer true for many embedded devices, especially ones that are consumer-oriented. Ideas from this research can therefore be used but have to be adapted to cope with an open world where many applications are executing alongside.

4.2 Modularized Virtual Machines

The virtual machine should be modularized, because it needs to be able to support various kinds of hardware. Monolithic virtual machines that need to be rewritten for every minor change in hardware are not economically feasible, given the rapid changing

[1] `http://domino.watson.ibm.com/comm/research.nsf/pages/r.plansoft.seminars.html`

[2] `http://erlang.org/faq/implementations.html`, section 8.4

[3] `http://techresearch.intel.com/articles/Tera-Scale/1421.htm`

[4] `http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html`

[5] `http://www.nvidia.com/page/handheld.html`

[6] `http://www.maclife.com/article/news/opencl_may_be_coming_iphone_near_you`

hardware. There are several attempts at building virtual machines at a higher level of abstraction. Some projects that interest us are for example:

- Jikes RVM(Alpern et al. 2000) and Squawk[7] are Java virtual machines written in Java.
- PyPy (Rigo and Pedroni 2006) is a Python virtual machine written in Python. The programmer is only required to provide an interpreter in a typeable subset of Python. A toolchain takes this as input and transforms it to a back-end language (such as C or Java), automatically adds a memory manager and even a JIT compiler. The advantage is that this is a high-level approach to build VMs, but on the other hand the programmer can only configure the VM for as much as is supported by the toolchain.
- Mate (Levis and Culler 2002) is a tailorable virtual machine specifically for sensor networks based on TinyOS(Gay et al. 2003). It can be tailored in its instruction set, which allows adding up to 8 custom instructions. Furthermore, the virtual machine can be configured to what primitives it supports (e.g. square root) and to what events it reacts (e.g. a timer, receiving of a packet, etc). However, since TinyOS is based on static memory allocation, Mate has no provisions for dynamic memory.

We are interested at taking any of these approaches and trying to replace some of the VM constituents by device- or even application-specific modules. A memory management module, for example, could be optimized to take advantage of a particular memory layout (e.g. a scratchpad memory of a particular size), reusing some of the research we have done at IMEC and described before. The goal is not to start every module from scratch. Rather, existing general purpose modules should be refined. This will require some fine-grained composition technology such as aspect-oriented programming or traits (Ducasse et al. 2006).

A composition engine can then compose the modules to make virtual machines for particular platforms, as shown in (Marr 2008).

4.3 Two-way Introspection and Optimization

The virtual machine should make it possible for applications to get information on what the virtual machine is doing and what resources it is using. The reason is that software can use this information for optimizations that are application-specific. For example, when the application is a scalable video codec, that has different quality options it can select from, what quality to show the user can depend on user preference (which is information locally available to the application) or on current platform load (which is information that should be obtainable through the VM).

Moreover the virtual machine itself should get information about the software it is currently executing, besides the actual bytecodes. The reason is that the virtual machine itself can do optimizations that transcend single applications. Take for example a componentized application that does video processing, where some components can be executed on either CPU or GPU. Depending on the current load and execution context the virtual machine could then assign the component to either GPU or CPU. The application itself cannot necessarily do this because it does not have enough information. Note that another possibility is to make this information available to the application component, like discussed before, in which case it can make this decision itself.

4.4 Adaptation

We can extend the introspection into reflective features where the VM adapts its workings based on the application it is executing. For example, the garbage collector module could be tuned to suit the applications that is currently executing, which is basically reconfiguring an internal part of the virtual machine. Even further we could replace entire modules of the virtual machine, for example replacing one garbage collector with another one or replacing the scheduler. This requires the virtual machine to monitor its own execution and adapt accordingly.

[7] http://research.sun.com/projects/squawk/squawk-rjvm.html

5. Conclusion

We are not virtual machine experts. We do have experience with writing code for battery-powered embedded devices, where C and assembly are used as development platforms and virtual machines do not seem to be making much progress. In this paper we try to outline some issues embedded developers currently face that do not seem to be addressed by virtual machines. Can virtual machines rise up to the challenges outlined here ?

References

B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalape no virtual machine. *IBM Syst. J.*, 39(1):211–238, 2000.

R. Baert, E. de Greef, and E. Brockmeyer. An automatic scratch pad memory management tool and mpeg-4 encoder case study. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 201–204, New York, NY, USA, 2008. ACM.

W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Design Automation Conference*, pages 684–689, June 2001.

S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.

D. Gay, M. Welsh, P. Levis, E. Brewer, R. von Behren, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *In Proceedings of Programming Language Design and Implementation (PLDI*, pages 1–11, 2003.

P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95. ACM, 2002.

S. Marr. Modularisierung virtueller maschinen. Master's thesis, University of Potsdam, 2008.

S. Marr, M. Haupt, S. Timbermont, B. Adams, T. D'Hondt, P. Costanza, and W. De Meuter. Virtual machine support for many-core architectures: Decoupling abstract from concrete concurrency models. In *Second International Workshop on Programming Languages Approaches to Concurrency and Communication-cEntric Software*, Electronic Proceedings in Theoretical Computer Science, York, UK, 2009.

J-Y. Mignolet and R. Wuyts. Embedded multiprocessor systems-on-chip programming. *IEEE Software*, 26(3):34–41, 2009.

Y. Neuvo. Cellular phones as embedded systems. In *Digest of Technical Papers of the IEEE Solid-State Circuits Conference (ISSCC)*, volume 1, pages 32–27, 2004.

A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953, New York, NY, USA, 2006. ACM.

C. H. van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1260–1265, 2009.

Ch. Wong, P. Yang, and F. Catthoor. Task concurrency management methodology to schedule the mpeg4 im1 player on a highly parallel processor platform. In *CODES*, 2001.

P. Yang, Ch. Wong, P. Marchal, Fr. Catthoor, D. Desmet, D.Verkest, and R. Lauwereins. Energy-aware runtime scheduling for embedded multiprocessor SOCs. *IEEE Design and Test of Computers*, 18(5):46–58, 2001.