

***VM-arm* : VirtualMHz for the ARM Architecture**

The Whitepaper

Copyright © 2005 Virtera

Doc #1002, Release 2

July 2005

Virtera, P.O. Box 2611, Santa Clara, CA 95055

1. Overview

VM-arm is a highly optimized and accurate simulation of the ARM architecture. It uses a sophisticated dynamic instruction recompiler, called the VirtualMHz Translation Engine, which dynamically converts ARM instructions into native host instruction sequences. This approach gives a significant speed-up over traditional simulation techniques.

The “VM” in **VM-arm** stands for VirtualMHz. This term refers to the clock frequency achieved for the simulated CPU. A simulation at 100.0 VirtualMHz will be equivalent in performance to a real ARM device clocked at 100.0 MHz. “Virtual” is used as a prefix to “MHz” to indicate that this is a simulation not a real device.

The main objectives are:

- Provide an accurate simulation of the ARM instruction set and system architecture.
- Very high simulation performance, capable of "real-time" ARM simulation (depending on the performance of the x86 host machine). For real-time use the VirtualMHz of the simulation must match the real MHz of the desired ARM device.
- Support a range of ARM software environments, including bare machine programs (running without an operating system) and full operating systems (such as ARM Linux).
- Provide a simulation of the Sharp Zaurus SL-5500 PDA.

This creates an ARM simulation environment that is ideal for ARM software development. Software can be prototyped and debugged before deployment on the real ARM system. The simulator provides rich debug facilities to support and accelerate the software development process.

An additional advantage of simulation is that a simulator can be extended to model modified ARM processors and systems before they reach production.

2. What Are The Advantages Of Using **VM-arm**?

VM-arm allows a PC running Linux to simulate an ARM platform. This is a pure software simulation and no additional hardware investments are required. We believe that **VM-arm** is the fastest whole-system software simulator for the ARM architecture in the industry. It also offers extensive features to support developers including debug features and plug-ins for additional device and platform models.

The major advantages of **VM-arm** are summarized below:

1. *Whole system simulation:* **VM-arm** is a whole-system simulator and it is possible to deploy software onto the **VM-arm** platform just like a real target ARM platform. This simulation naturally includes an ARM micro-processor including its Memory Management Unit (MMU) and on-chip devices, but can also be extended to include many of the devices and system features of the whole ARM platform. This can include execution from the very first instruction after reset, a complex operating system (e.g.

Linux), device drivers, software stacks, middleware and applications.

2. *Real-time simulation*: The performance of a **VM-arm** simulator can match the real-time characteristics of the target ARM processor. Performance in the range of 50 to 200 VirtualMHz is typical, though the performance characteristics will vary according to the capabilities of the host system. **VM-arm** has a clock synchronization mode allowing simulated time to be synchronized to real wall-clock time. This ensures that the simulation does not run too fast and smoothes out simulator performance over time.
3. *Accuracy*: The accuracy and completeness of the simulator is clearly very important for correct execution of guest programs. **VM-arm** simulates the whole ARM CPU architecture including the full instruction set and co-processors. It supports the ARM MMU and exceptions. This allows all code to be run under simulation including boot-strap and the operating system.
4. *Debugging*: The Developer's version of **VM-arm** will support powerful debug mechanisms to allow software bugs to be quickly analyzed and root caused. This will include the ability to plug the simulation into standard debug environments so that usage of the simulator can be identical to usage of a hardware development platform. Additionally, **VM-arm** can provide sophisticated trace and debug capabilities built directly into the software simulation.
5. *Extendibility*: The Developer's version of **VM-arm** will provide a plug-in API so that additional device and platform models can be added to the simulation. This ensures that **VM-arm** can be extended to include models of platform-specific features. Additional features will include configuration files for defining the platform, and a scripting language interface for controlling the simulator.
6. *Time to market*: **VM-arm** can accelerate the time to market for an ARM-based product by allowing software development to start far ahead of hardware availability. Additionally, software developers can be more productive using a simulation environment because of the sophisticated debug facilities leading to additional time to market savings.
7. *Cost*: Development boards and platforms are not cheap. Sometimes they can be more expensive than a low-end PC. **VM-arm** can convert a standard desk-top PC into a sophisticated ARM development environment. In combination with open source cross compilers (such as the Gnu compiler collection and libraries) and open source operating systems (such as Linux), the entry cost to embedded software development has never been so low!

3. Why Simulate The ARM Architecture?

The ARM architecture is everywhere, literally:

- In your cell phone.
- In your PDA or handheld games console.
- In your car.
- In your digital camera.
- In your home network.
- In your iPod or MP3 player.

In terms of volume and design wins the ARM architecture is probably the number one 32-bit micro-processor architecture. Most ARM applications are in the embedded space, typically in consumer devices.

Software Complexity

The complexity of ARM devices is increasing rapidly. Many hardware capabilities are being integrated with the processor giving large System-On-Chip (SOC) devices. The software that is being shipped on ARM devices is becoming very sophisticated, and is also growing in complexity. The software stack may include low-level code, operating systems, drivers, communications stacks, middleware and applications. In many respects the software running on an embedded system is becoming similar to the software in a general-purpose computer, both in quantity and capability. The number of software developers for the ARM architecture is also large giving a significant market for software tools and development systems.

The ARM software developer will typically prototype the software on a regular PC, a standard ARM development board, or on the target ARM platform itself. There are limitations to these approaches:

- *A PC is not an ARM device:* A PC provides a sophisticated programming and development environment but it is completely different to a real ARM device or platform in many important ways. A PC will have a much faster processor, more memory, a completely different architecture and different i/o devices. Only high-level prototyping is typically possible on a PC leaving a large amount of rework when moving to a real ARM target.
- *ARM development is expensive:* Development boards are relatively expensive, in some cases as much money as a low-end PC! A standard ARM development board may not match the target ARM platform in terms of processor speed, available memory or i/o capabilities. Custom ARM development boards require prototyping and bring-up. Debugging software on a development board may be difficult. Often memory is scarce and the programming environment is unfriendly.

- *Time to market:* In some cases it is not possible to use the target ARM platform - often it does not even exist at the point when work is started on the software. It is highly desirable to overlap hardware design and development with software design and development to improve time to market. Usually this means starting work on the software before the hardware is even ready. In the case of an ARM custom processor or ASIC (which is very common for high volume consumer products), the target ARM chip itself may not exist. Alternatively, maybe the board design or platform bring-up has not been completed. Even if the target ARM platform is available, it may be very unfriendly to software development due to a lack of debug facilities or i/o capabilities.

Simulation Benefits

A software simulator can address these problems by providing an accurate, fast simulation of the target ARM platform early in the design cycle. Additionally, the simulator offers advanced integrated debug capabilities that are simply not possible or practical with a device in the real world. This can include complete tracing of instruction execution or data accesses, or complex instrumentation to detect problems early and help determine their root cause.

Software simulation can be very cost effective, requiring licensing of a software package rather than acquisition of expensive development hardware. *VM-arm* is designed to be the fastest whole-system ARM simulator in the industry with the objective of simulating complete ARM devices and systems in real-time. This allows the simulator to be used as a direct replacement for the target hardware system.

4. Why Use A PC Host?

The reasons for simulating on the PC platform are simple: the PC is ubiquitous, affordable and provides an extraordinary amount of computing power for simulations. In fact, it is the performance ratio between a typical PC and a typical embedded ARM device, that makes real-time software simulation possible.

Clock Rate

The clock rate of a typical modern PC is between 1.0 GHz and 4.0 GHz (almost).

An ARM processor in an embedded application typically has a clock rate between 50 MHz and 400 MHz. This is a factor of 10 to 20 slower than the PC clock range. Some ARM devices have even lower clock rates (e.g. for very low power), though a few are higher. In terms of volume the lower clock rates are much more common.

Memory System

The memory system in an ARM device is much less powerful than a PC. A PC has large

on-chip caches and might have single-channel or dual-channel PC3200 DDR memory. A typical ARM will have a much narrower memory interface (e.g. only 16 or 32 bits of data) and slower single data rate SDRAM or SRAM. Also, an ARM micro-processor has very much smaller data and instruction caches, and no L2 cache. Finally, an ARM platform will often use FLASH memory, and this is very much slower than RAM. If programs are executed-in-place in the FLASH memory, then execution will be particularly slow.

Additionally, a typical modern PC has comparatively huge amounts of memory (512 MB to 2 GB), while An ARM device might only have 32 to 64MB.

Software Environment

A PC has a familiar software development environment, lots of disk space and will often be used to host a cross-development environment to develop the ARM software. A general-purpose computer is an ideal platform for hosting a simulation.

5. The Simulated Hardware Platform

The current release of *VM-arm* models the following features of the Intel StrongARM SA-1100 processor and Zaurus PDA platform:

<i>SA-1100 Processor Features</i>	<i>Zaurus Platform Features</i>
ARM instruction set, little-endian operation only.	LCD screen: 320 x 240 x 8-bit color. Touch screen: emulated by clicking on the LCD window with the mouse.
Exceptions, aborts, interrupts.	Keyboard and other buttons.
Co-processor 15 (memory management unit).	A serial port can be connected to the terminal window in which the simulator is run.
Memory bus: interfaces to ROM, FLASH, SRAM and DRAM.	A serial port can be connected to the host for networking using PPP.
System-on-a-chip features: memory controller, LCD controller, serial comms module, UARTs (serial port drivers), 28 general-purpose I/O pins, real-time clock, interrupt controller, timers, serial ports.	64MB of RAM and 32MB of ROM. CompactFlash memory card (maps a host file as the memory card contents). CompactFlash ethernet network card (connects into the host's network).

Additional ARM devices and platforms will be supported in future releases.

6. The VirtualMHz Translation Engine

VM-arm executes ARM programs using both interpretive simulation and translation.

Interpretive Simulation

Most simulators implement the guest system by *interpreting* the instructions supported by the guest CPU. The simulator works on one guest instruction at a time. Each instruction is decoded and then executed by calling code that simulates the effects of that instruction on the state of a machine.

Translation

A more efficient technique of simulating the guest system is to *translate* the instruction sequences executed by the guest processor into equivalent instruction sequences that can be directly executed by the host processor. The translation process decodes the guest instruction sequences, and translates them one-by-one into host instruction sequences. Many optimizations are possible during this translation process, and many of the overheads of interpreted simulation can be reduced or entirely eliminated. The translation process is generally much slower than the interpretation process. However, the translated sequences can be cached and then reused if they are required again. Since many programs have high code reuse (e.g. Due to loops or reuse of procedures), the cost of translation can often be amortized over many uses giving a significant speed-up over interpretation.

The translation process is more expensive than simply executing an instruction by interpretation, particularly if aggressive optimization techniques are used. However, the translated sequence itself runs much more quickly than interpreting the same set of instructions. There are many reasons for this speed-up. A few of these are:

- There is no need to fetch or decode the ARM instruction.
- The instruction can be translated into an in-line sequence of x86 instructions with no need to make a call via a dispatch table.
- The x86 instruction sequence can be highly optimized.
- Many decision points (and branches) are removed because the generated x86 sequence is completely customized to the ARM instruction and not general-purpose.
- Optimizations between ARM instructions can be made.

The amount of optimization performance on the translated sequences is a trade-off. Conservative optimizations is relatively quick, and produces sequences with a moderate speed improvement. The aggressive optimization can be much slower. A typical approach is to generate an intermediate instruction representation following by "global" optimization at block or fragment level, rather than just instruction level. The pay-off is that an aggressively optimized sequence can be even faster than the conservatively optimized sequence.

Performance Characteristics

To put this context, assume that the performance characteristics are as follows:

- 10 host cycles to execute 1 ARM instruction assuming that it has already been translated using aggressive optimization.
- 20 host cycles to execute 1 ARM instruction assuming that it has already been translated using conservative optimization.
- 100 host cycles to interpret 1 ARM instruction (i.e. no translation).
- 1,000 host cycles to translate 1 ARM instruction with conservative optimization.
- 2,000 host cycles to translate 1 ARM instruction with aggressive optimization.

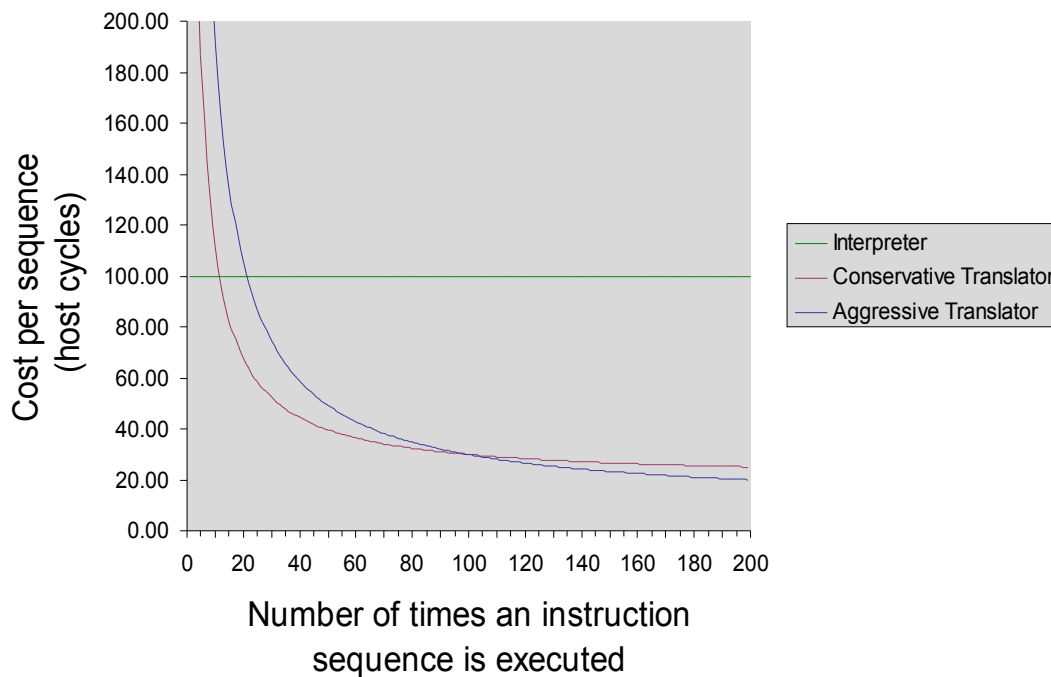
These numbers are approximate and not based on any measurements or data. However, they are useful “rules of thumb” and should be of the right order.

If a particular sequence of ARM instructions is executed only once in the life-time of the program then it is best to execute it by interpretation. If it is executed many times, however, it is worthwhile translating the sequence and applying conservative or aggressive optimization. If there is very heavy reuse of the instruction sequence, so that the translation cost can be amortized over many uses, then the speed-up from aggressive optimization approaches 10 times the interpreter, and 5 times for conservative optimization.

The decision on whether to interpret or to translate, and whether to optimize conservatively or aggressively, is critical to performance. The simulator must make good predictions about the future likelihood and frequency of a block being re-executed in order to find good candidates for translation. Fortunately, it turns out that past behavior is typically an excellent guide to the future patterns of instructions executed by a program. This is the same program property that is exploited by instruction caches and branch predictors in a CPU design. *VM-arm* therefore gathers profile data on the past execution characteristics of the code in order to make good decisions about when to translate and how to optimize.

The following chart shows how the cost of executing an instruction sequence varies according to the number of times that this instruction sequence is executed. It shows the affect of amortizing the translation cost across many uses of a cached translation. The interpreter has the same cost for an instruction sequence regardless of how many times it is executed.

Interpretation vs. Translation



Using the cross-over points in this graph:

- Interpretive simulation should be used for an instruction sequence that is executed 12 or less times.
- Translation with conservative optimization should be used for an instruction sequence that is executed between 13 and 100 times.
- Translation with aggressive optimization should be used for an instruction sequence that is executed more than 100 times.

These cross-over points are very dependent on the assumptions. In particular, the cross-over point from conservative to aggressive optimization is high because of the assumed high cost of aggressive optimization and the assumed 2x resultant speed-up. With other assumptions, such as lower cost of aggressive optimization or higher performance gain, the cross-over point will be lower and may be more attractive.

These numbers are also optimistic in the sense that they assume accurate prediction of the number of occurrences of a sequence ahead of time. In practice, the simulator will choose to interpret initially when there is no information on which to predict future usage. Translation (at an appropriate level of optimization) will only be used later if it becomes clear that the instructions are likely to be heavily reused. However, at this point the simulator has already “paid” the cost for several interpretations of the instructions before

any translation is done. This has the effect of raising the cross-over point at which translation is worthwhile as the earlier costs of interpretation have to be recovered and amortized.

Note: currently **VM-arm** performs conservative optimizations only. It is estimated that performance can be improved by a factor of 2 for very frequently executed ARM sequences using an aggressive optimization strategy. The resultant effect on overall performance will be less than a factor of 2 since not all blocks are executed so frequently, but will still be a significant improvement.

There are other factors that have a significant effect on the implementation and performance of the translation mechanism and the translation caches:

- *Unpredictable branches*: Branches, especially indirect branches and calls by function pointer, cause control to flow to unpredictable instruction addresses.
- *Writes to code pages*: This occurs in self-modifying code and code that loads other code. Although true self-modifying code is relatively rare, it is common to write code pages. For example, when an operating system loads a program into memory or fixes up load-time or run-time relocations.
- *System effects*: context switches, cache/TLB coherency sequences, exceptions and interrupts.

The effect of these is often to flush part or all of the translation cache causing previously cached translations to be discarded. Often they will then be retranslated in order to execute future code. A good translator needs appropriate strategies to deal with these situations without overly complex or expensive algorithms, yet prolonging the lifetime of entries in the translation cache.

7. Timing Model

The simulator uses a simple but effective timing model. It uses instruction timings that are a simplification of the cycle timings of the SA-1100. Loads and stores are charged an extra 2 cycles to provide a simple model of data cache misses, while exceptions cost an extra 6 cycles. The timing model is a compromise between timing accuracy and speed of simulation.

Free-running Mode

In free-running mode, **VM-arm** runs freely without regard to wall-clock time. Timing of the model will be set to match the requested MHz value (either specified or the default). The main effect of the requested clock frequency is to determine the clock frequency for the timers and clocks within the simulation. This determines, for example, the rate at which clock/timer interrupts will be generated in the simulation. If the requested clock frequency is set lower than the actual clock frequency achieved by **VM-arm**, then “real-time” in the simulation will pass more *quickly* than wall-clock time. Conversely, if the

requested clock frequency is set higher than the actual clock frequency achieved by *VM-arm*, then “real-time” in the simulation will pass more *slowly* than wall-clock time.

Clock Synchronization Mode

In clock synchronization mode, *VM-arm* attempts to synchronize its achieved wall-clock execution rate to match the requested MHz value (either specified or the default). If this can be achieved then *VM-arm* will run such that “real-time” in the simulation matches wall-clock time, giving fully real-time behavior. This means that a real-time clock in the simulation will match wall-clock time. Another advantage is that real-time games in the simulation will play at the right speed! This mechanism would also be particularly important for audio emulation; however, audio emulation is currently not implemented.

Clock synchronization can only be accomplished if the requested MHz rate is less than or equal to the achieved MHz rate. This allows *VM-arm* to insert small delays, as required, to keep the achieved rate of execution synchronized to the requested rate of execution. These delays are achieved by the operating system's sleep system call. This gives free CPU cycles for other processes to use.

If the achieved MHz rate is lower than the requested MHz rate, then *VM-arm* cannot keep up with real-time. No delays will be inserted and the behavior is essentially the same as free-running mode.

8. Performance Data

Performance data is given in the following table:

<i>Benchmark</i>	<i>Interpreted Execution</i>	<i>Translated Execution</i>
Dhrystone 2.1 benchmark 10,000,000 loops	19.8534 VirtualMHz	216.4951 VirtualMHz
First 4 billion instructions of Zaurus Linux boot (this takes the Zaurus from power-on to its desktop)	19.2097 VirtualMHz	71.1465 VirtualMHz

This was for a reference machine consisting of:

- Dell Dimension 4600 desk-top.
- 2.666 GHz Intel Pentium 4 with 512 KB of L2 cache.
- 1.5 GB of DDR memory.
- Fedora Core 3 Linux, kernel 2.6.9-1.681_FC3.

This is essentially a commodity desk-top PC, though it has a large amount of memory. A high-end machine will have a significantly higher clock speed and larger L2 cache, and will be as much as 50% faster than this reference machine.

The simulator was set for a desired clock frequency of 103.0 MHz. Clock synchronization was disabled so that the simulation would be free-running (i.e. not throttled back to the requested frequency).

This requested frequency was exceeded in the Dhrystone 2.1 case giving 216.4951 VirtualMHz. This is faster than the 206 MHz of the SA-1100 processor! Note that this is a small benchmark run on a bare machine environment with the MMU enabled, but no operating system. The translator is able to translate all of the Dhrystone main loop and all of it stays resident in the translation cache. This results in a very high performance.

The Zaurus Linux boot benchmark is a much more demanding test. In particular, many Unix processes are created and destroyed (e.g. during shell script processing such as the init process), and the translation mechanism is much more heavily exercised. An achieved performance of 71.1465 VirtualMHz is about one third of the real Sharp Zaurus SL-5500 which has a 206 MHz SA-1100 processor. However, the time from power-on to desktop is much closer (about 50% slower): about 60 seconds for the *VM-arm* model and about 40 seconds for the Sharp Zaurus. The reason for this performance disparity is that the Sharp Zaurus has very slow memory compared with *VM-arm*. The Sharp Zaurus uses DRAM and FLASH to hold data and code, and the average memory access latency is much larger (especially for FLASH) than the *VM-arm* timing model. Additionally instruction and data cache hits rates will likely be worse than the *VM-arm* timing model. In practice, execution of Zaurus programs under *VM-arm* is almost as fast as the Sharp Zaurus SL-5500.

Also note that the interpreter is itself very efficient and highly optimized. The performance of the interpreter is sufficient for many software development tasks, though the translator is needed to give “real-time” simulation. For example, Zaurus pac-man runs in real-time with the translator but not with the interpreter.

9. What Software Can Be Used With VM-arm?

VM-arm is a simulator, and will run bare machine ARM programs as well as various flavors of Linux. The simulator can also support other software environments that run on ARM processors.

Here are some common scenarios:

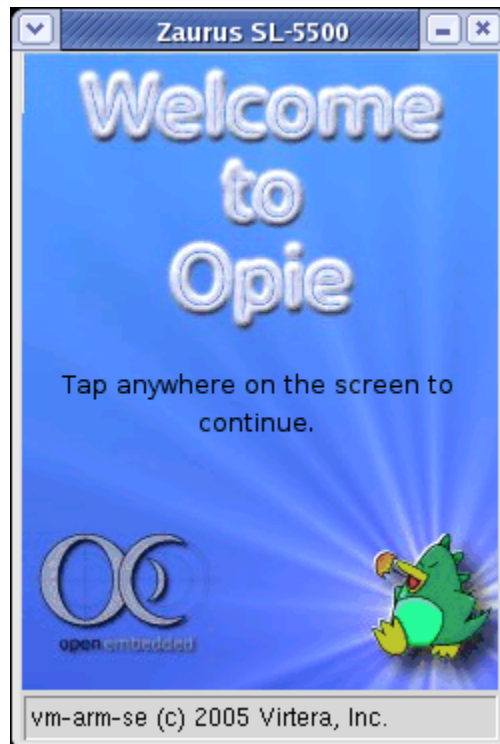
- Some users may already have developed ARM software for use with ARM hardware or development boards. This software can be run on *VM-arm* using a built-in monitor for host input/output.
- The GNU tool chain can be used to build programs to run on the simulator. There are two main variants:
 - `arm-elf`: the GNU tools used to build programs that run directly on the

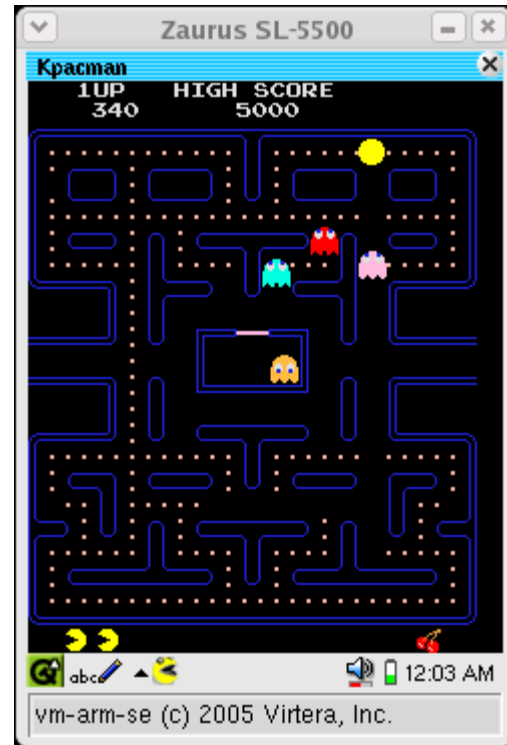
bare machine (with no operating system)

- arm-linux : the GNU tools used to build programs that run under the Linux operating system.
- Various Linux operating systems can be downloaded or built to run on the simulator:
 - 2.6 series Linux kernels can be downloaded from www.kernel.org and will run out of the box on the simulator (with appropriate configuration choices).
 - The OpenZaurus project provides an open-source Linux-based operating system and graphical environment for the Zaurus hardware platform. These can be downloaded from www.openzaurus.org and can be run directly on the simulator without modification.

10.Show Me The Screenshots!

The following pictures are screen shots from running the OpenZaurus kernel on the simulator. This includes the OPIE welcome screen, various screenshots of application icons and settings, and Pac-man! These are screenshots directly from *VM-arm* and were sampled using the GIMP.





11.Future Plans

Further performance enhancements are planned. A development version exceeds 25MHz for the interpreter and 100 MHz for the dynamic translator on the Linux boot benchmark. Aggressive optimization of translated code sequences will yield up to a further factor of 2 beyond this. It is expected that a consistent 200 MHz can be sustained for complex codes (e.g. Linux boot and applications) with further optimization work on the simulator and a faster host system.

The *VM-arm* product currently supports both Intel and AMD PC platforms. The simulator also runs as a 32-bit program on the 64-bit x86 architecture. There is approximately a 40% improvement in simulator performance from a 2.666 GHz Pentium 4 to a 2.2 GHz Athlon 64 3400+. Currently the simulator itself does not take advantage of the 64-bit architecture. The availability of 16 x 64-bit registers and improved ABI calling conventions will result in a further performance improvement.

Support for other Zaurus or other PDA platforms is also planned. These platforms are used as the main development platform for a number of reasons:

- They are readily available and affordable.
- A large amount of off-the-shelf software is available for testing. This is particularly true for the Zaurus where almost all software is open source and freely available.
- They have a large number of interesting i/o devices.

Support for a more recent Zaurus platform will provide simulation of Intel XScale PXA-series ARM processors, larger LCD screens and larger memory configurations. Finally, there are many additional features planned for the Developer Edition of VM-arm.

12.Sounds Interesting ... Tell Me More!

Additional information is available at the Virtera web site: <http://www.virtera.com>. The web site includes the **VM-arm** documentation, FAQ and bulletin board. A demo version of the **VM-arm SE** software is available as a free download.

13.Terminology

Simulation vs. emulation: The terms simulator and emulator are commonly used to describe the implementation of a computer system (e.g. A micro-processor, memory and input/output mechanism) in software. These terms are used somewhat interchangeably and it is not easy to differentiate them. Perhaps, an emulator aims to implement the "visible" interfaces of a computer system, while a simulator models the internal mechanism. Alternatively, perhaps an emulator models a complete computer system while a simulator is more focussed on the processor. However, these definitions are quite arbitrary and largely depend on your point of view. In this document **VM-arm** is referred to as a simulator.

Host and Guest: The host is the computer on which the **VM-arm** software is run. Currently, the only supported host systems are 32-bit x86 based personal computers running Linux. The guest is the computer which is simulated by the **VM-arm** software. Currently, the supported guest system is the StrongARM SA-1100 processor, optionally with peripherals compatible with the Sharp Zaurus SL-5500 PDA.

Source and Target: It is sometimes convenient to consider the interpreter or translator as mapping instructions from a *source* processor (this is the guest) into a *target* processor (this is the host). These terms can be somewhat confusing when used in general – e.g. some may consider the ARM as being the target of the simulation. Therefore, these terms are generally not used and only specifically when discussing the interpretation or translation process.

14.Legal

This document is copyright © 2005 Virtera. All rights reserved.

This document may be electronically redistributed subject to the following conditions:

- The document must be distributed in its entirety and without modification. The document may not be converted into any other file format. It must be kept in the original PDF format without change. Virtera's copyright notices must not be removed.
- Permission is only granted to redistribute this document where there is no charge to the

recipient of the document. Permission is not granted to redistribute this document for any kind of fee.

- Permission is not granted to redistribute this document in any other format or media. Paper redistribution of this document is not permitted, only electronic redistribution in the original PDF format.

15.Trademarks

AMD, Athlon and Opteron are trademarks or registered trademarks of Advanced Micro Devices, Inc. ARM, Thumb and StrongARM are trademarks or registered trademarks of ARM Limited. Dell and Dell Dimension are trademarks or registered trademarks of Dell, Inc. Intel, Pentium 4, Xeon and XScale are trademarks or registered trademarks of Intel Corporation. Mandrake is a registered trademark of MandrakeSoft, SA Corporation. Linux is a registered trademark of Linus Torvalds. Microsoft and Windows are trademarks or registered trademarks of Microsoft Corporation. Red Hat, RPM and Fedora are trademarks or registered trademarks of Red Hat, Inc. SUSE is a registered trademark of SUSE LINUX AG, a Novell business. Sharp and Zaurus are trademarks or registered trademarks of Sharp.

Other products named in this document may be trademarks or registered trademarks of their respective owners.

The *VM-arm* software was developed using the Fedora Core operating system, available from <http://www.redhat.com/fedora/>. The *VM-arm* software is built with GNU compilers and the GNU tools, and dynamically linked against GNU libraries. More information about the GNU Project can be found at <http://www.gnu.org>. The *VM-arm* software runs on the Linux kernel, see <http://www.kernel.org/>. The GNU compilers, libraries and tools can be used to cross-compile programs into ARM executables to run under *VM-arm*. Also, the Linux kernel can be built to run under *VM-arm*.

The *VM-arm* software uses cryptographic features from the OpenSSL Toolkit. The OpenSSL Toolkit itself is not included in the *VM-arm* software distribution and *VM-arm* itself contains no source code from the OpenSSL Toolkit. Dynamic libraries for the OpenSSL Toolkit must be provided by the host operating system. More information about the OpenSSL Toolkit can be found at <http://www.openssl.org/>.

The *VM-arm* software uses wxWidgets which is an open source C++ GUI framework. The wxWidgets libraries are not included in the *VM-arm* software distribution and *VM-arm* itself contains no source code from wxWidgets. Dynamic libraries for wxWidgets must be provided by the host operating system. More information about wxWidgets can be found at <http://www.wxwidgets.org>.

Software from the OpenZaurus Linux distribution can be run on *VM-arm*, and this can be obtained from <http://www.openzaurus.org>. The OpenEmbedded core library and tools are used to build OpenZaurus. More information about OpenEmbedded can be found at <http://www.openembedded.org>.