

THE MAGIC OF VIRTUALIZED SYSTEMS DEVELOPMENT

OCTOBER 2009

David Beal

WWW.VIRTUTECH.COM

WHAT IS VSD?

In the 1999 movie (US) “The Matrix”, the main character “Neo” found that in a virtual world, he could do things that would otherwise be impossible. One memorable scene showed him dodging bullets by bending back and forth at hyper-speed.

Similarly, *Virtualized Systems Development*[™] (VSD[™]) enables “impossible” approaches to the development of electronic systems. The benefits extend across nearly every phase of product life cycle – all the way from definition, to development and finally deployment. With VSD, a hands-on investigation of candidate system designs is possible, hardware and software teams can collaborate in new ways from the outset of a project, bugs can be found more quickly than ever, system integration can start much earlier, and customer support and training programs are more effective and easily delivered.

WHY ADOPT VSD?

In just a few years, electronic systems have become exponentially more complex. Now, even comparatively simple designs include multiple processors, a mixture of CPU types, DSPs, ASICs, FPGAs and other devices. Complementing the diverse combinations of hardware, today’s systems employ a variety of operating systems and application stacks that until recently would not have been combined within a single product or solution.

Unfortunately however, as these systems have grown in complexity, the development tools and processes that were refined when single processors and basic client/server architectures were the rule, have not kept pace. As a result, today’s system developers are challenged to find new ways to define system architectures, develop and integrate millions of lines of code, and to deploy such complex systems. They must do this in ways that reduce risk and schedule while simultaneously resulting in a higher quality, easier to support and maintain product.

BENEFITS OF VSD

The introduction of VSD into the work flow of an electronic project can benefit everyone who is involved in product definition, development and deployment. These benefits are summarized by the table below according to the position type and role.

Engineering	
CTO & system architects	Hands-on “what-if” analysis of new system architectures
Software and systems engineers	Speed your development with new development and debug techniques Work in your own office on your own schedule, not at the ESD lab during your 2am time slot
Integration and test teams	Uncover problems early through continuous iterative systems integration Configure, integrate, and test full systems and their full software stacks Automate system testing Pick up exactly where you left off on the previous day Easily communicate bugs to developers
Managers	
CFO	Release your product to market faster Save more than 30% on software development costs Improve the use of resources: staff, hardware, instruments Eliminate the cost of providing hardware to globally distributed teams
Engineering manager	Reduce schedule, resource and feature/performance risks Efficiently spread work among global engineering teams Get the hardware and software teams working together at the outset of the project Demonstrate key architecture decisions before implemented in hardware
Product managers	Free software progress from hardware availability Reduce development schedule and cost

Marketing	
Technical marketing	Create laptop-hosted demonstrations of early product visions or the final product
Software ecosystem partners & product managers	Deliver third party software (e.g. OS ports, application stacks) when hardware becomes available – not months later
Product Support	
Training sessions	Create encapsulated, laptop portable and web-deliverable hands-on training sessions
Customer-specific configuration	Support every customer-specific configuration without a lab full of equipment
Bug replication	Duplicate the customer’s precise configuration Replicate bug reports by sharing checkpoint files with the customer
Bug correction	Allow any engineer to quickly duplicate bugs that have been previously observed
Identify source of bug: silicon vendor code, 3 rd party code or end-user code	Use identically configured systems to confirm that every code element is functioning (or not)
Quality Assurance	
Early integration and test start	Catch bugs earlier and correct them with less effort
Guaranteed bug replication	Easily duplicate bugs found by the integration or test team (on a virtual platform)
Corner case testing	Perform corner case testing of any hardware, software or system parameter
“Impossible” testing	Force the hardware to behave incorrectly to trigger “impossible” bugs

HOW MUCH CAN VSD REDUCE COST AND SCHEDULE?

By using VSD, the amount of time and money that can be saved across the product life-cycle is considerable. These savings come from both first order factors (e.g. faster debug) and from second order factors (e.g. earlier product release).

First Order Factors

First order factors directly affect the calculation of project cost and schedule. When a VSD-based approach is used, the most influential first order factors are improvements to product definition, development, debug, integration, and support.

Many different methods have been used to estimate the cost of software projects. One of these approaches is Cocomo II¹. This model is based on detailed statistical analysis of years of data from real-world software development projects and it yields cost, schedule and resource estimates for specific project phases including architecture investigation, software development, integration and test. Because VSD notably affects these same phases, Cocomo II is especially well suited to estimate the savings between VSD-based and traditional development approaches.

In order to calculate a project's cost, schedule and resource estimates, Cocomo uses detailed project-specific data such as lines of code, programming language, percentage of software re-use, and library usage. It also uses subjective data such as personnel skills, development tools, team collaboration, and many other parameters.

For the purpose of estimating the savings from a VSD-based approach, we have defined a small software project consisting of:

- RTOS with 100k SLOC, 90% original
- Four device drivers @ 5k SLOC
- Two device drivers @ 1k SLOC

¹ COCOMO is being used by several large companies such as IBM, Lockheed Martin, EDS, Hughes, Microsoft, Motorola, Xerox and dozens of others, see http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html

- Four Small applications: 10k SLOC
- Two Medium applications: 100k SLOC
- Two Protocol: 75% original, 10k SLOC
- Four Libraries: 100% re-used, 20k SLOC
- Security: 100% re-used, 50k SLOC
- Total = 295k SLOC

Using the example model above, we ran three separate estimates: baseline, conservative and aggressive. Each of the three models assumed different degrees of improvement to the subjective areas identified by Cocomo: analyst capability, risk resolution, design team flexibility, team cohesion, process maturity, and development tools. Note that the objective parameters (e.g. lines of code, programming language, percentage of software re-use, and library usage) remain unchanged and as a result had no affect on the project estimates regardless of development approach.

The baseline estimate assumes a traditional development approach and uses default Cocomo II values for each of the subjective parameters (e.g. Personnel skills, development tools, etc.). The conservative and aggressive estimates both assume a VSD-based approach with measurable improvement to those same subjective factors. The conservative case assumes one level of improvement (compared to baseline) and the aggressive case assumes two levels of improvement in these subjective areas.

The table below summarizes the calculated cost and schedule savings of both the conservative and aggressive cases when compared against the baseline, non-VSD case.

Conservative Estimates		Aggressive Estimates	
Cost savings	Schedule savings	Cost savings	Schedule savings
36%	18%	60%	28%

Projected Cost Savings VSD vs. Traditional Development Methods

Because most projects are orders of magnitude larger than our small example above, we also explored cost estimate linearity based on project size. We found that as the lines of code grew by 2x, the improvement to cost and schedule was between one and two percent better than that shown by the

smaller project. We concluded that the estimates above conservatively reflect savings that would be gained on larger projects.

Second Order Factors

Second order cost factors result from those positive or negative changes to schedule, features or quality. For example, if a product is delivered early, then the positive impact of early release can be substantial in terms of product sales and market leadership. On the other hand, releasing a product with a significant defect might result in a corporate black-eye and loss of customers, the cost of which may not be known for months or years after product release.

Often, the true cost (or revenue) of such second order factors can only be determined by your organization based on knowledge of your product, its market and your competitors.

WHAT MAKES A VIRTUAL PLATFORM SO POWERFUL?

“Virtual Platforms” are best described as functional models of physical hardware. They are used as the target for software development. A virtual platform can represent a basic board with a processor and memory, or it can be a complete system made up of network-connected boards, chassis and racks.

The accuracy and fidelity of the model is such that the target software is unable to distinguish the virtual platform from physical hardware; it runs the same binaries and behaves exactly like physical hardware. When these high fidelity virtual platforms are combined with a feature-rich simulation environment, developers can define, develop, deploy and integrate target-specific firmware, operating system kernel, device driver code, application and communication stacks even while the hardware design and production progresses in parallel.

The use of virtual platforms provides many benefits to the product life cycle resulting from traits of the virtual platform, or from the development techniques that can be employed on those virtual platforms.

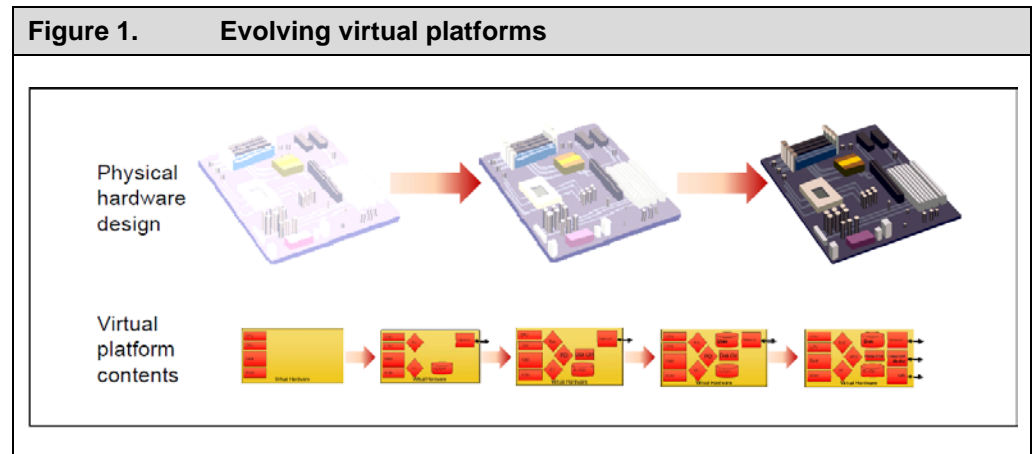
Virtual System Flexibility

A virtual platform will often evolve incrementally, aiding software developers all the way from the model’s origins as a basic platform to the

final full-system model. When using virtual platforms, it is important to remember that a complete model of the system is no more important to specific software development tasks than a CD player is required to check the brakes on your car!

Often the functionality or even existence of a hardware device (e.g. IO, ASICs, or a specific memory implementation) – is not required for a specific software task. In these cases, that device can be simply omitted, stubbed out, or replaced by off-the-shelf models that provide similar functionality.

For example, boot code that initializes some device by writing register values can be developed, debugged and tested on a model consisting of only the CPU, memory, and the device registers directly used by the code. All functionality, beyond providing the read/write register for initialization, can be deferred until needed for OS porting and device driver development.



Virtual platforms are unique in their ability to function even when they do not model every aspect of the full hardware. This trait allows software development to progress even as the virtual platform and the hardware design itself evolves.

Executable Specifications

With VSD, hardware and software teams are able to experiment with new system architectures by creating and networking together several virtual platforms. This virtual system can then be used to run real software loads for the purpose of prototyping and design concept testing during the architecture or systems definition phase of a project.

When used in this manner, the virtual platform becomes a living *executable specification* that can be used by everyone - systems engineers, hardware and software teams, marketing teams and sales people. Like traditional requirements documents and specifications, the virtual platform model(s) can be saved and archived where they will serve as a template for future product designs.

The use of an executable specification encourages hardware and software teams to work together from the very start of a project. Now, fundamental design problems can be detected and resolved, long before they migrate into physical designs. Traditional development approaches (e.g. without VSD) might only discover these issues during systems integration where they become expensive and problematic to correct.

Simplified Build Environments

Because the virtual platform runs the same software binaries that will be run by the physical system, there is no need for a special cross compiling, simulation or development builds. This aspect of VSD can significantly reduce the number of code build variants for a project, reducing maintenance costs and the risk of errors from inconsistent builds.

Re-usable Technology Assets

Today, even the smallest consumer device may be comprised of a diverse combination of processor cores and devices. This hardware complexity requires that the underlying simulation infrastructure will allow building-block creation of these real-world combinations of hardware. With VSD, every individual simulation component and platform model (processor cores, devices, custom ASICs, application-specific FPGAs, board or network models) becomes a reusable company asset. For example, once an IP block or ASIC has been modeled, that model can be reused in any future system designs.

Just as hardware designs connect physical components, model components are combined in virtual platforms. By continually expanding and integrating the model library, the benefits of VSD for developers and projects, expands too.

VIRTUAL: BETTER THAN REAL?

Virtual platforms enable the impossible. VSD enables developers to monitor every bit and every register within the hardware, freeze execution of the whole system, inject any hardware error, take system checkpoints, precisely repeat every operation from run to run, automate every step of the system's operation, and even to run the system in reverse!

Easy to Share, Re-Use and Deploy

The simulation infrastructure should provide host-independence, so that any virtual platform is able to run on any host computer while maintaining the exact same target software behavior. Host independence guarantees that virtual platforms will work exactly the same way in Seoul as they do in San Jose, or after your developers have gone from two to eight core desktop systems.

Full System Stop

Unlike physical hardware, virtual platforms and complete virtual systems can be completely and synchronously stopped. With virtual platforms, there is no wind-down delay ("skid"). All processor cores stop with a single command – not from separately issued debugger commands as used in physical systems. With virtual platforms, the stop includes not just processor cores but also peripheral devices and even data in-flight on networks and buses. In this frozen state, developers have full access and visibility to all hardware and software variables. They can set new software and hardware breakpoints and then resume normal execution – all as if the platform had never stopped. VSD also allows the whole system to be single-stepped by taking advantage of this full system stop combined with full hardware visibility.

Checkpoints/Snapshots

How often have software developers wished to return to a previous point in time so that they can continue their work from the point where they left off during the previous session? With physical hardware, the system must be re-run through the same sequence of events in order to arrive at a point that approximates the original point in the platform's execution.

Developers need the ability to capture, save, and restore the complete system state so that the operations can be resumed – at some point in the future – at exactly the point where they were left off. This "checkpoint" capability

allows an engineer to effortlessly continue work after a long weekend, system crash or even years later when trying to extend legacy software².

Repeatability

How many times have engineers detected a bug, only to be unable to repeat (and correct) it? The primary reason for this lies in the inherent chaotic behavior and variable timing of physical hardware. On physical hardware, as we all know, repeating a bug may take weeks or months as developers work to isolate the precise set of parameters that will force the system into a faulting state.

Virtual platforms, on the other hand, are fully repeatable. If a bug is seen once, it can be replicated any number of times with ease. How? Simply load a pre-bug system checkpoint and then run the system forward. Repeatability ensures that the bug will be hit at exactly the same time and after exactly the same actions for every run that begins from the common checkpoint.

Reverse Execution

“If I only knew then, what I know today!” Hindsight is 20:20 – a problem is far easier to identify and correct after it has happened.

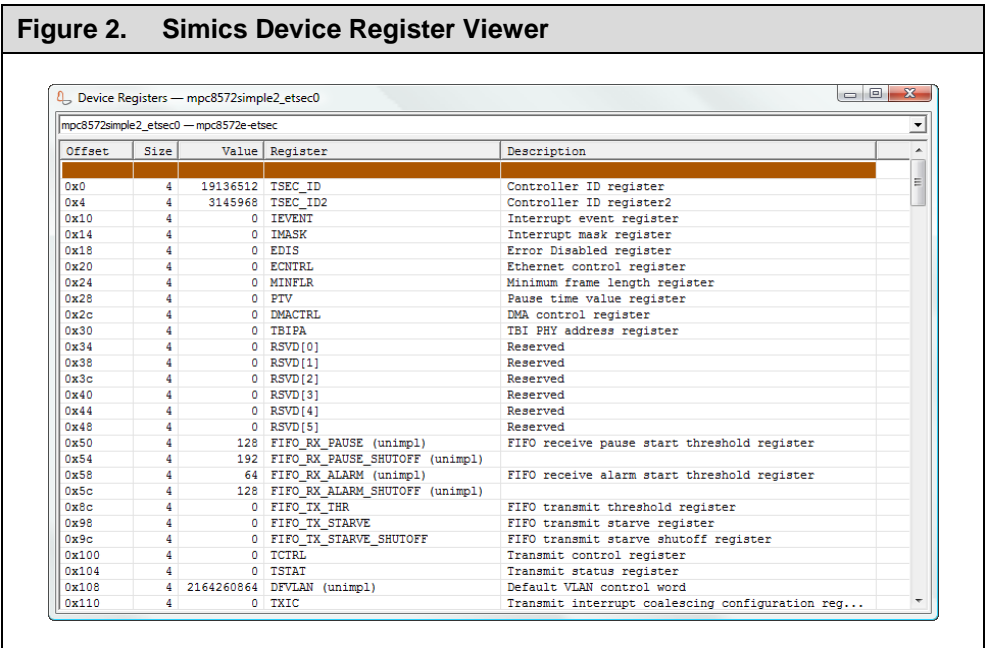
Reverse execution complements system repeatability by allowing developers to run their systems *backwards* beginning at a point *after* the problem occurs. Debugging in reverse is straight forward because, even while the system proceeds backwards to a point before the problem occurs, it continues to stop at every software and hardware breakpoint along the way. This capability alone has saved developers months of effort on difficult to find system bugs.

System Visibility, Control and Fault Injection

With physical hardware, the developer’s access to specific registers is limited to those that the SoC designer chose to make visible. This often includes most registers on the CPU and a select set of registers on the MMU and supporting IO devices. Unfortunately, some OS kernel and driver software must access devices or registers that are simply not visible (or debuggable) on physical hardware.

² See http://www.virtutech.com/whitepapers/simics_checkpointing.html for more on checkpointing in Simics

On a virtual platform, every bit, byte and register on the system can be both inspected and modified to provide faster development and higher code quality. Figure 2 shows how one can inspect the registers of a hardware device. This visibility enables developers to debug every line of software and view its full interaction with every hardware component, easing for example, development of software controlling deep internal devices such as a TLB. Now, with the ability to access *all* hardware registers, test teams can inject faults for corner-case scenarios that are impossible to verify with physical hardware.



Scripting and Automation

As we've seen, VSD provides complete virtual platform visibility and controllability. This control allows complex system operations to be automated (or scripted) for precise configuration, control and duplication. Any scripted actions can be defined in terms of time "at precisely cycle 41691156, an interrupt from the Ethernet controller will occur", and/or it can be defined in relation to other elements in the virtual system "at precisely the moment that the landing gear lock indicator illuminates, corrupt the contents of register r5 and r6." Scripting can also be used to perform interaction with the system such as reading the serial console output before entering appropriate commands and text according to programmed instructions.

This ability to automate the system to such a fine level of detail enables the detection and duplication of bugs, system corner case testing, integration and test, creation of advanced training scenarios and sales demonstrations.

Flexibility

With a virtual platform, it is simple to change the latency, speed, or size of memory. It's easy to add another processor onto the platform, add a new platform into the system, or add new network interfaces. Now, developers can experiment with more configurations than using physical hardware, both to find configuration-related bugs and to explore performance scalability as hardware changes – long before physical hardware is changed.

Scalability

Full-system models can reach sizes that include hundreds or thousands of individual modeled components. Unless the simulation infrastructure supports scaling, these large model sizes can quickly reduce the performance to a point where it becomes unusable by software developers.

In order to fully exploit VSD, the simulation infrastructure must be able to exploit the power of today's multicore and network-connected servers.

WILL VSD HELP YOUR ORGANIZATION?

Because VSD introduces new high-level approaches to product development, several adopters have evaluated their own needs through four steps:

1. Identify product development process challenges
2. Identify risk and cost
3. Identify where VSD might reduce those challenges
4. Identify cost, schedule and quality benefits of VSD

Identify Product Life Cycle Challenges

The first step in analyzing whether your organization can benefit from VSD is to list each of the challenges facing your development teams, product and engineering manager, marketing and support teams.

Issue	Challenges
TEAM CHALLENGES	
Geographically dispersed development teams may not operate as efficiently as desired	Are your global engineering teams difficult to coordinate and manage? Are engineering efforts done in relative isolation from other teams? How effective is it for one team to request assistance, or to hand-off problems to another team?
Hardware and software teams may not collaborate well	Do your hardware and software teams play the blame game: "It's a ___ware problem, talk to them!!"
HARDWARE CHALLENGES	
Developers have limited ways to test "what-if" scenarios for different CPU or cache configurations, board topologies, and system architectures	Complex systems introduce many parameters that affect the operation of the overall system. Does a paper analysis of these parameters meet all of your design needs?

Issue	Challenges
<p>Availability of development systems, JTAGs, or Logic Analyzers for all of your developers</p>	<p>How stable and reliable is your alpha and beta hardware?</p> <p>Do you have enough hardware platforms for all of your software developers?</p> <p>Do you have enough power supplies, JTAG debuggers and logic analyzers to support developers?</p> <p>How closely does the prototype hardware match the final product? What software changes need to be accommodated as a result?</p>
<p>DEVELOPMENT CHALLENGES</p>	
<p>Software development is contingent on access to boards, subsystems or new CPUs</p>	<p>Does progress on OS porting, application stacks, systems software, and inter-platform communications require access to hardware?</p>
<p>Some bugs take weeks or months to repeat and then find.</p>	<p>How often has your product shipment been delayed while trying to resolve critical bugs?</p> <p>How many engineers can practically debug a single problem at one time?</p>
<p>Developers and testers have scheduled time slots on prototype systems</p>	<p>Do engineers have access to development and production systems as necessary to efficiently complete their work?</p> <p>How much does it cost your organization when 24 hour shifts are needed for system integration and test?</p>
<p>Corner case testing of hardware and software across the parametric limits is incomplete.</p>	<p>Every hardware attribute has a min/max and typical specification.</p> <p>Can you prove that the system functions under specific combinations of hardware characteristic (power, timing, latency, etc).?</p>
<p>MARKETING CHALLENGES</p>	
<p>Paper specification and PowerPoint presentations may be unable to collect all required customer requirements and feedback</p>	<p>How good is your ability to vet candidate designs and features with customers and to demonstrate prototype and intermediate designs?</p>
<p>Customized product demonstrations are impossible to provide prior to system completion</p>	<p>Would it be helpful to demonstrate your product's capabilities and features before you deliver?</p>

Issue	Challenges
INTEGRATION CHALLENGES	
Critical design problems are found during system integration	What is the cost and delay to fix 11 th hour problems? Because system integration always falls in the critical path, it can determine the success (or failure) of the product.
Delivery delays from 3 rd parties affect your project's schedule	Does your product rely on elements from 3rd parties? What is the risk when your project's critical path falls outside of your own organization?
SUPPORT CHALLENGES	
Support teams struggle to repeat customer problems	How many resources are spent trying to duplicate a customer problem?
Modular, training courses that are configurable, quick and easy to deliver are very difficult to produce	Can you perform user-training in traditional class rooms on commodity equipment? Do you equip and maintain a separate hardware lab just for customer training?
Support for customer-specific system setups requires dedicated hardware labs and hardware configuration	Can you easily support your customer's diverse system configurations? How much hardware and effort is required to reproduce customer-specific configurations?
PRODUCT DEVELOPMENT LIFECYCLE CHALLENGES	
Any project's critical path can benefit from more parallelism	A project's critical path includes a number of serial tasks: hardware development → hardware (alpha) manufacture → low level software development → application software → system software → system integration. Delays at any of these points can affect development schedule and cost.

Identify Risk and (Opportunity) Cost

Based on the challenges identified in the first step, managers often analyze the risk and cost of such challenges. Costs should include direct costs as well as opportunity costs. Key areas that have been identified by VSD adopters include:

Issue	Description	Risk	Side Effects
Market Test	Inability to test that the product includes the proper features and capabilities	Poor product definition	Unmarketable or unsellable product \$\$\$ Millions lost
Development Problems	Bug detection and correction, poor hardware/software team coordination, limited access to hardware, late start to systems integration	Schedule slips Late arrival to market,	Loss of market leadership, contract penalties \$\$\$ Millions lost
Quality Issues	Critical system or unit bugs	Flawed product Unmarketable product	High support costs Corporate “black eye” \$\$\$ Millions lost
Support Issues	Difficulties in duplicating errors, limited access to customer equipment	Poor customer satisfaction Poorly performing equipment	Lost customers Corporate “black eye” High support costs
3 rd Party (Ecosystem) delay	Key complementing hardware or software elements are not available	Release delays Adoption & mass market delays	Incomplete solution \$\$\$ Millions lost
Sales Engagement	Inability to demonstrate that a specific configuration will meet customer need	Lost sales	\$\$ Tens or Hundreds of thousands lost

IDENTIFY WHERE VSD CAN BENEFIT THE PRODUCT LIFE CYCLE

Virtualized Systems Development complements traditional hardware-centric work flows by bringing new methodologies, work flows and techniques for product definition, development and deployment.

The adoption of VSD is an evolving process. As VSD is used, more models are created, developers become more familiar with what can be done and the resultant value and benefit of VSD grows too.

Product Definition Phase

The virtual platform enables marketing, hardware and software teams to leave behind a paper-trail of documents and to define the system architecture in a hands-on, dynamic and demonstrable manner. Some customers have even used virtual platforms to demonstrate their proposed solution as a part of the project bid process.

Example: Architecture Investigation and Design

Virtual platforms are especially valuable in the system design process when the end-product is derivative of an existing product for which models already exist. Here, the flexibility and scalability of virtual platforms allows new system configurations to be quickly created and tested. Typical use cases include:

- Migration of a function from software running on a CPU to an independent accelerator (e.g. ASIC, FPGA, PLCD)
- Investigating a change in the number of platforms (e.g. configuring a system with three routers and six switches or two routers and seven switches)
- Verification of an application's correct operation after porting from a single-threaded platform, to a multi-threaded platform.

Product Development Phase

Often, during a product's development phase, progress may be challenged by issues around hardware availability and readiness, schedules, debugging, inter-system communications, and systems integration. Each of these issues can be mitigated with virtual platforms because they are binary compatible with the physical hardware, they can be used for everything from early boot code development all the way through integration and deployment.

Not only is a virtual platform a good *replacement* for hardware for software development, but it's support for "magic" features that just are not possible on physical hardware (e.g. full-system stop, run to run repeatability, full visibility & control, reverse execution, checkpointing, and scripting/automation) combine to make it a superior development platform. In-fact, after using VSD for just a matter of weeks, many software developers prefer to use virtual platforms for their efforts.

Example: Progressive System Integration with Virtual Platforms

While the use of VSD for OS porting and application stacks might be obvious, it also provides large benefits to system integration and testing. Physical hardware requires that each subsystem be completed before it can communicate and interact with other subsystems. VSD however requires only that the white-box functionality of each subsystem is duplicated at the level needed to "fool" the rest of the system. Often, the effort needed to provide this level of white-box functionality – just enough so that other subsystems can operate in their normal manner – is orders of magnitude less than the effort to develop the missing subsystem's software.

By wisely using the incremental model approach, VSD allows system integration to begin solely on virtual platforms, expanding to a combination of virtual and physical hardware, and finally to fully physical hardware as those components become available.

Example: Debugging Systems with Reverse Execution

There are three phases to software debugging: Repeating the bug, isolating the bug and fixing the bug. When working with physical platforms, the successful completion of these steps requires a high degree of developer skill and experience. VSD on the other hand dramatically trivializes repeating the bug, dramatically eases bug isolation and speeds correction of the bug. We have several examples of software developers who had struggled for months

to repeat and isolate bugs on physical hardware, only to find them in days with VSD.

Repeating the Bug

To repeat the bug on physical hardware, the system or application may have to be restarted hundreds or thousands of times, each time using a new set of input parameters, data streams or operator actions that are hoped to provoke the bug.

Virtual Platforms are different. They operate in a virtual world where every data stream and IO parameter can be scripted or captured and provided again and again on all subsequent runs. As a result, the system always starts from the same point thus ensuring that all system functions, timing, activity and results are also duplicated. This ensures that a bug that has occurred once can be trivially repeated simply by loading an existing checkpoint (snapshot file) and re-running. By using checkpoint files that ensure a bug will occur, developers can easily collaborate with worldwide team members, all on a single bug.

For developers who desire the randomness of physical hardware, virtual platforms can be provided with different “seed” values. This will provoke different system responses - just like physical hardware, while still maintaining full repeatability for any given set of seed values. New seed values are used to provoke different system responses for corner case testing. These seeds can be provided using manual, automatic or metric driven verification techniques.

Isolating the Bug

Once the bug can be reliably repeated, the developer must find the source of the bug. Traditional hardware-centric debug methods require an iterative approach where an initial breakpoint will be set and the system run until stopped. After the system stops, standard debugging tools are used to view the CPU registers and stacks – all with the goal of finding out whether the problem occurred yet or not. If the problem has occurred, another breakpoint located before the previous breakpoint is set, and the application or system is re-run. If the problem has not yet occurred, a break point is set after the previous breakpoint and the application re-run. Using this technique, developers eventually are able to find the precise offending line(s) of source code.

Debug procedures changes completely with a virtual platform that can run in reverse. Now, developers have the benefit of knowing when the problem occurred and they can backward step through the code – observing all hardware and software breakpoints along the way – to quickly find the precise point of origination. Such an approach does away with the iterative “guess/stop/re-start” technique required by physical hardware and instead takes a reverse linear path that begins after the problem has occurred and ends at the point where the bug begins.

Fixing the Bug

Often, the effort to fix a bug, whose source has been isolated, is much less than the preceding effort to duplicate the bug and to isolate its source. VSD does not provide tools that will help the developer to *write* the proper code. Developers do however benefit from those same capabilities for checkpointing, reverse execution and run-to-run repeatability that were helpful in repeating and isolating the bug.

Product Deployment Phase

Due to the effort required to create models, if a VSD-based approach has not been used earlier in the product life cycle, it is often not practical to introduce VSD as late as the product deployment phase. However, when virtual platforms have been created during the product definition and/or development phases, they can be used effectively to encapsulate customer-specific configurations, to provide easy to deliver hands-on training programs, and to aid in customer support.

Example: Encapsulating Customer-Specific Configurations

Most complex systems can be configured in different ways according to the customer-specific need. Some customers might need fifteen routers, twenty switches and ten gateways. Other customers might have just five routers, four switches and one gateway. Although each of these deliveries uses the same components, the quantity and distribution of subsystems can vary. While good for customer scalability and customization, this scenario causes a support problem for the product supplier who often finds that they must reconfigure a dedicated lab of equipment for each specific support scenario.

Virtual platforms can be quickly reconfigured and scaled – all without any physical hardware present. Once a specific configuration has been produced, it is trivial to restore and run that same configuration in a matter of seconds.

HOW CAN ORGANIZATIONS IMPLEMENT VSD?

VSD introduces some fundamental process change into existing product life cycle work flows. As a result, its adoption is not an overnight process. It may take some time for the engineering, marketing and support teams to *fully* realize what can be done, and how it should be done in order to gain the benefits of VSD. Instead, the adoption of VSD should be an evolving process. As it is used, more models are created, developers become more familiar with what can be done and the resultant value and benefit of VSD grows too.

Identify Long-Term Organizational Goals for VSD

The checklist below can be used to help prioritize some of your long-term organizational goals and hopes for VSD. It can also be used to help assess the features and capabilities of candidate solutions.

Issue	Need/Priority
Product Definition Tasks	
Need for customer feedback	
Demonstrating very early prototypes	
Fast prototyping of evolutionary releases	
Product Development Tasks	
Early solution (hardware+software) delivery	
Virtual labs for customer-specific configurations	
Software development independence (from hardware)	
Ease of system scaling	
Global engineering team efficiency	
Bug detection and correction speed	
HW/SW team problem solving abilities	
HW/SW team collaboration	
Efficiency and use of resources (developers, hardware, equipment)	
Product Deployment Tasks	
Hands-on demonstrations	

Product releases to 3rd party eco-system partners

Lab-less product training programs

Support for customer-specific equipment configurations

Creation and Retention of Corporate IP and Workflow

Creation of virtual platforms

Re-use of 3rd party device models

Using external consultants to create models

Creation of in-house model building skills

Integration with existing development tools and work flows

Model-Driven Architecture (MDA) tools

Hardware design (EDA/ESL) tools

IDE and Debug tools

Hardware, Software and HW/SW verification systems

Build and version control systems

Identify Short-Term Opportunities for VSD

As VSD represents a significant change to the way that engineering, marketing and sales teams have done business, it can be challenging to introduce into an organization. Consequently, it is best to introduce VSD into projects which can immediately benefit from its features and capabilities.

Identify VSD-candidate projects: Often, the projects that offer the easiest path for the introduction of VSD are within the system architecture or software development phases.

However, projects in the deployment phase can also be easily supported if they are based on commodity hardware for which an off-the-shelf model exists or if those models can be easily produced. If the hardware used is sufficiently expensive or hard to configure, creating new VSD models will be profitable even for quite custom hardware setups.

Identify the intersection of physical & virtual platforms: Because it takes some training and effort to create models, those projects that can benefit immediately from VSD will have a good correlation between the physical hardware platform and the available virtual platforms or component models.

Identify problematic projects or programs: Projects that have progressed far into the software development and even deployment phases have used VSD to capture and resolve bugs that had evaded resolution for months. VSD also provides developers and testers with increased access to hardware so that they can put more resources towards problem solving.

SELECTING THE SIMULATION-MODEL INFRASTRUCTURE

The simulation-model infrastructure is that software which manages and runs virtual platforms. Some might consider it to be a simulation engine or the simulator itself. Many features and characteristics of this infrastructure are critical to enabling the full benefits of VSD.

High Level Features

The model infrastructure must contain key high-level system features including:

Fast performance – the system must be fast enough to run the target software fast enough to satisfy software developers who are used to running

software on physical hardware. Speed is one thing that differentiates a VSD platform from those simulation tools provided by the electronic design automation (EDA) industry. Although EDA tools are extremely accurate from a hardware perspective and they can be used to develop low level initialization and test code, they are too slow to be practical for OS, application or systems software

Scalability and High Fidelity (accuracy) – Virtual models must be able to scale from a simple register model to one that precisely duplicates the functionality of the physical hardware component must be supported

Host Scalable – In order to retain adequate performance, the model infrastructure must be capable of utilizing multiple hosts for work sharing on large models

Extensive Off-the-Shelf Model Selection – Given the complexity of model development, the selected solution must not only support the ability to run complex, mixed-architecture systems, but it should also provide a range of off-the-shelf models across the various CPU, DSP, and network options. These models can be run stand-alone, or they can be used as the basis for custom model development

Technology IP Re-use

Adoption of VSD will suffer if the solution is too difficult to use or if it becomes too difficult to enable different engineering teams to collaborate with their resultant model IP

Import of standard model and register languages – e.g. The SPIRIT Consortium's IP-XACT and SystemC TLM-2.0-using models.

Support for multiple modeling languages – provides flexibility and inter-operational capability.

Multiple levels of abstraction – the level of required functionality for most model components will vary according to the software task to be completed. The model infrastructure must support everything from omission or simplification of devices all the way up to providing full functionality.

Model scalability – the ability to create larger, more complex models from a set of smaller subcomponent models – engineering teams must be able to re-use models developed by other teams.

Interface with physical hardware– the ability to interface virtual platforms with physical hardware via standard communication interfaces or RTL emulators expands use-cases of virtual platforms

Easy creation of large models from smaller models – the use of basic scripting language to create complex models – modeling some components can be difficult. The creation of complex platforms that incorporate these models should be much easier

Comprehensive Silicon/Electronic System Support

In order to enjoy the full benefit of VSD, a number of capabilities for the modeling of real-world systems are required:

Modular and flexible – allow new complex models to be created from smaller subcomponents

Multicore and ASIC support – complex systems use complex processors

Mixed processor architectures – e.g. Power Architecture+ DSP + x86 + MIPS

Mixed Operating System models – e.g. Hypervisor + Linux + RTOS + bare metal

Buses and communications networks – e.g. Ethernet, PCI, PCI-Express, RapidIO, MIL-STD-1553, ARINC 429, SpaceWire, FireWire, USB, ATM, serial.

Virtual to Physical world connection – allows virtual platforms to communicate with physical platforms.

Features for the Software Developer

Features that must be available to developers in order to expedite their development efforts include:

No software development tool changes – developers should be able to use the same software development tools (e.g., compilers, linkers, debuggers and IDEs,) and development processes that are used with physical hardware.

Whole system freeze – allows debugging in a system-inactive state

Repeatability – allows easy replication of bugs and simulation results across teams, time, and developers, eases collaboration on specific software problems by multiple developers

Checkpoints and Snapshots – saves and restores the full system state for future use, save time by creating a particular hardware-software setup once for a large group of developers

Advanced debugging – reverse execution, hardware and software breakpoints

Full hardware visibility and control – allows developers to see registers and data that would be invisible on physical hardware

Script driven automation – for testing and validation

BUILDING MODELS AND VIRTUAL PLATFORMS FOR VSD

Just like physical hardware, high fidelity models can be large and complex. When an ASIC, SoC or FPGA model is properly designed, each can be used far beyond the narrow use case for which they had been initially created. When it is poorly designed, it can limit the features and benefits available with VSD.

In order to ensure maximum performance and re-use of virtual models, model builders should follow some standard practices.

Do not over-model

Engineers should evaluate the system from a software functional perspective in order to determine which registers and functions should be modeled. For example, if the software will never read register M, it is not necessary to model register M. If register O provides the result of a BIST (Built-In-Selftest), it is only necessary to model the result *register* so that it can be read by software (you can easily set the BIST value to pass or fail to see how software reacts). There is no need to model any of the BIST function itself.

Ensure Checkpointing and Reversibility

By ensuring that every model can be checkpointed, your engineers will be able to easily revert to a known system state. Engineering teams will be able to collaborate by sharing that system state with each other. With some

additional care, models can also support reverse execution, thus providing a key benefit to software developers during the debug phase.

Use Common Modeling Interfaces

The simulation infrastructure should offer standard APIs for communications between different model components. When possible, each modeled component, no matter how large or small, should adhere to these standards. The goal is to make sure that all models created can be seamlessly integrated into larger platforms, racks or systems.

Create Stand-Alone Models

By creating fairly fine-grained stand-alone models that are designed to be combined into larger models, developers ensure that their technology IP is reusable across many different projects and products in the future. It also ensures that the model itself is easier to debug. For example, it is far easier to debug an ASIC that has been broken up into component devices (or functions) than it is to debug one that has been written as a large monolithic model.

SUMMARY

VSD complements traditional hardware-centric development to offer very large benefits in terms of developer efficiency, software quality, and reduction in delivery schedules. These benefits can only be exploited through the combination of a fully-featured simulation infrastructure with a full set of models and with the ability to create custom models. As more models, use cases and developer experience build, the benefit, value and ROI of VSD adoption builds too.

For more information on Virtualized Systems Development and how it is enabled by Virtutech Simics®, please visit www.virtutech.com.

CONTACT INFORMATION

North and South America sales_americas@virtutech.com	Europe, Middle-East, Africa sales_emea@virtutech.com
Asia-Pacific sales_apac@virtutech.com	Japan sales_japan@virtutech.com
http://www.virtutech.com	

© Copyright 2009 Virtutech, Inc. All Rights Reserved.

TRADEMARKS. Virtutech, Simics, Hindsight, and the logos thereof, are trademarks or registered trademarks of Virtutech, Inc. and/or its subsidiaries, in the United States and/or other countries.

THIS PUBLICATION IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. VIRTUTECH MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Virtutech, Inc., 2001 Gateway Place, Suite 201E, San Jose, CA 95110, USA