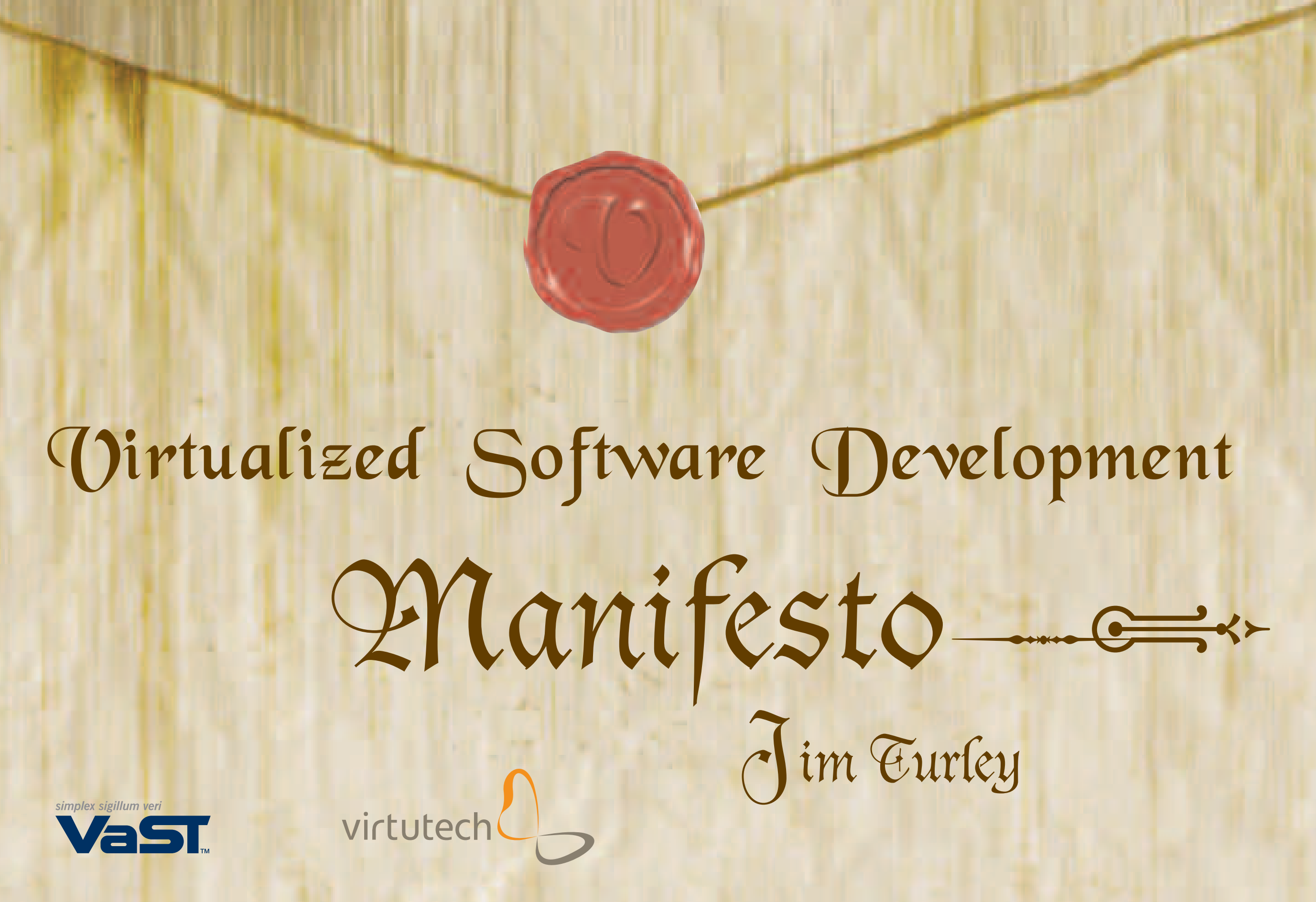


Virtualized Software Development Manifesto

Jim Turley

Oh please, not another paradigm shift. Hardly a day goes by that we're not inundated with earnest pleas to change the way we do engineering, programming, quality control, or management. The industry press is filled with the cacophony of strident voices telling us about new "inflection points" or "game-changing technology" and "seamless functionality methodology platform solutions." Or something like that.

May the gods of engineering preserve us from their followers. For most engineers (i.e., the ones with jobs), we just want to get on with our work. Sure, the new developments in tools and languages are interesting but they're also kind of abstract and irrelevant. They might be useful for the next project or the project after that, but for the time being they're a distraction.

For now, the average programmer wants to finish coding, debug the code, fix a few undocumented features along the way, and ship the damn thing. Lather, rinse, repeat. When the hardware works and the code runs, life is good. When the hardware isn't ready, or is buggy, or isn't really representative of the final product, life becomes a bit tougher – but hey, that's all part of the game.

Or is it?

Flaky hardware is a given in all engineering labs. The hardware guys (and gals) are trying to bring up prototypes while the software geeks (and nerds) are trying to write code for the nascent hardware. You can't write software for nonexistent hardware and you can't debug the hardware without diagnostic software. A familiar drama plays out as programmers wait for hardware and the hardware engineers wait for working code. It's the push me-pull you of the product-development zoo. Each side tugs and prods the other but eventually, step by step, the hardware becomes stable enough for the software and vice versa.

Theory versus Practice

Let's say – just for the sake of argument – that it didn't have to be this way; that it could be done just a little bit differently. Let's assume – hypothetically, of course – that programmers didn't need hardware to write hardware-specific code. As an entertaining little thought experiment, let's assume for the moment that hardware isn't actually necessary to build a product. (Some marketing experts have suggested that this is already the case: that hardware is generic and software defines the product. Cell phones, DVD players, and iPods are good examples of products built on generic hardware that are differentiated solely by their software. But that's an argument for another day.)

In this theoretical world any programmer could write any type of code – from operating systems to device drivers to interrupt handlers to application programs – without reliable hardware or, indeed, any hardware at all. Any programmer could create software without hardware underneath it. And not just write the code but also debug it, test it, profile it, fix it, update it, anything. (Obviously, anyone can write lines of code without hardware; the trick is running and testing it.) In short, we'd assume "virtual" hardware resources were available in lieu of the real thing.

There's plenty of precedent for this. It's not a revolutionary, or even a particularly original, idea. Operating systems handle "virtual memory" all the time, swapping code in and out of physical memory and adjusting pointers to make it appear that there's more memory than there really is. We "virtualize" device drivers so that disks all behave the same even though hard drives, floppies, "thumb drives," and CD-ROMs have almost nothing in common.

So we're standing on solid theoretical ground here. Any hardware can be "cloned" in software, and with enough fidelity that it's functionally identical. Like the classic Turing machine, a user can't tell a simulation from the real thing. Paradoxically, the software doppelganger is sometimes even faster than the hardware it emulates, as Transmeta discovered with its Pentium-compatible Crusoe chip. DEC's x86-translation software could execute some hardware instructions faster than Intel's own microprocessor could. The list goes on, but the point is that software surrogates have a long and distinguished history.

So what's the point? Companies use software virtualization for commercial gain, to get access to existing software, to simplify software development, and to make programs more portable. Still others do it to buy time – and that leads us back to our engineering lab.

A Standup Fight or Another Bug Hunt

Virtualized hardware could (theoretically, remember) deliver eleven major advantages to programmers. We'll call these our Eleven-Point Manifesto of Virtualized Software Development.
It will allow programmers to:

- Develop software before the target hardware is available
- Explore alternative hardware configurations
- Definitively isolate software bugs from hardware bugs
- Improve confidence in software-only testing and results
- Retain expertise with current languages, tools, and software
- Improve debug visibility and controllability compared to actual hardware
- Free up actual hardware for hardware debugging
- Isolate the effects of silicon or board revisions from the debug process
- Stay ahead of the "complexity curve" for new systems
- Speed debug time by eliminating hardware configurations
- Improve product quality by isolating the source of bugs more quickly

Starting with the first point, it's clear that virtualized hardware allows a programmer to begin writing before any actual target hardware is available. That in itself is valuable to the average development team. Or at least, to their boss. If software development could even start a few weeks earlier – never mind proceeding with greater productivity – without hardware, that's a serious financial incentive. And it

eliminates a common source of friction between hardware and software engineers: sharing the supply of working hardware.

As a nice side effect, chip makers can supply virtual models to their prospective customers, software partners, and allied vendors. Then, when the first chips are available, the third-party support and infrastructure are already there and early customers will have a head start on their software and support logic.

While we're on the subject of early prototypes, virtualized software wouldn't care what the hardware looked like, so it would have the nice side effect of encouraging developers to explore different configurations. It's much easier to mock up a software model of a four-processor system than it is to design and build one. It's cheaper, too. Want to know how adding more Ethernet channels would affect throughput? How cutting memory capacity would change performance? How switching processors would aid performance? It's much easier to find these answers through software modeling than through brute-forcing the hardware.

When the software is developed independent of the hardware, the inevitable bugs in the code are just that – software bugs. There are no hardware bugs because there's no hardware. Software models aren't intermittent like hardware glitches are; they're not flaky or heat-sensitive. (Granted, the software model of the hardware can be incorrect, but it can't be unreliable.) Virtualized platforms aren't sensitive to power fluctuations, crosstalk, or component tolerances. They're not back-ordered from the PCB manufacturer. They don't change from batch to batch. They don't have blue wires on the back that come loose at inopportune moments. And they're not constantly being swapped out for newer, updated boards or borrowed for testing. In short, the virtualized hardware is a stable, reliable foundation on which programmers can build their own code, not teeter precariously on someone else's bug-infested platform.

When software bugs are isolated from hardware bugs, confidence increases. Everyone can be satisfied that, yes, the software has been thoroughly and adequately tested. There's no question that hardware race conditions or peculiarities in some peripheral chip are masking a latent bug. If the code works on the model, the code actually works. Any bugs that abruptly appear when the code is run on real hardware are, by implication, hardware-related bugs. The software's already been tested; the debugging job is already half done.

One of the charms of virtual hardware is that it makes no demands on the software tool chain. No new languages, no special compilers, no run-time virtual machines. Software development proceeds apace, just as before. The programmers are happy because they're doing what they've always done, with all the tools they've done it with. Survey data shows that programmers are far more loyal to their compilers than their processors. They'd swap hardware in a heartbeat but will defend their choice of software tools to the end.

Debugging is equal parts art and science. The science is knowing how to fix flaws; the art is knowing where to look. Development boards bristle with probes, wires, switches, and LEDs in an effort to instrument as much as possible and make the invisible visible. It's a constant battle between what the programmer needs to see and what the hardware allows him to see. Virtual hardware makes that entire class of problem disappear. A virtual processor, board, or system is infinitely visible. There is no register, bus, or memory location that isn't testable, changeable, and verifiable. A whole host of tricks and kludges that programmers have developed over the years goes out the window, the one case where software techniques do change with virtualized hardware.

When hardware development boards are available, they can stay with the hardware engineers. Obviously, they'll need to test and debug their own early revisions while the programmers are toiling away in the adjoining lab. But now they don't have to share their precious development boards. The hardware can stay with the hardware group, speeding up their debug cycle. This is just as useful in later stages when the hardware gets updated or upgraded: the revised hardware can stay with the engineers while the programmers work with their updated virtual model.

Part of the excitement of early-stage debugging comes when the chips are just as new and untested as the rest of the product. Whether it's a newly released microprocessor or a new FPGA design, the chips often go through several revisions just like the board or system. Isolating chip quirks from hardware idiosyncrasies shouldn't be the programmers' job. They'll have their hands full debugging their own code without wondering why it worked yesterday but doesn't work today. Gee, the hardware looks the same...

It's no secret that embedded systems get more complicated over time, not less. That means more hardware and more software, and more development time poured into both of them. Bringing up a new model of next year's system is far easier and quicker than actually building it. Virtualized software allows the development team to get a head start on next year's product design or next year's technological leap. Multi-core processors, new peripherals, and next-generation interfaces hold no terror for the development team armed with virtual models.

Oftentimes a new product will be sold in many configurations. Bigger hard disk, smaller memory size, different expansion capabilities, what have you. Naturally, both the hardware and software teams need to test all of these. Just as frequently, the software will be intended to work on all of them, automatically adapting to features that are or aren't enabled. That requires testing on different hardware configurations, and that usually requires a visit from a technician. Whether it's changing jumper settings, adding or removing a mezzanine card, swapping out controllers, or rearranging little green wires it's something the software team often isn't equipped to do. Once again, eliminating the hardware eliminates the problems of hardware configuration. Different models of the target can be tested as easily as throwing a software switch.

Finally, it all comes down to quality. Not the sexiest of characteristics, but one that's hurt more than a few reputable companies. Quality can't be "tested-in," it has to be designed-in. Simply scrapping failed products does nothing to eliminate the root cause of the failure – and it doesn't help the bottom line, either. Fixing failures means first finding them, and finding them is vastly easier and quicker when the hardware and software can be separated from one another.

More of What You Want

A nice theory, this. We've divorced software development from its long-time partner hardware development. We've endowed the former with almost magical properties of speed, reliability, and X-ray vision to see inside of chips. We've made programming (or at least, debugging) quicker and more rewarding, and all without changing the programmers' tools one bit. But was this all just a pointless fools' errand?

Of course not; white papers always have happy endings. There are a handful of honest and upright companies that do this very thing. They're not always well-understood or recognized, but they're very real and their customers are quite deliriously happy with the result.

Virtutech and VaST Systems are two such companies, with their Simics, CoMET, and METeor products, all of which help engineers virtualize the design of software and hardware. It's not a new way to design hardware (although there are plenty of ways to do that) but a new way of developing software independent of hardware. It's at once pedestrian and revolutionary; both familiar and exceptional. Virtualized software development, or pre-hardware programming, makes no demands on the programming staff and actually lessens the load on the hardware staff. The two factions become more independent of one another, and more able to pursue their own goals separately. The hardware will eventually meet the software, of course, but it won't be a shotgun wedding... more of a long courtship. And the product of that union will likely be healthier and better adjusted than ever before.

Examples Good and Bad

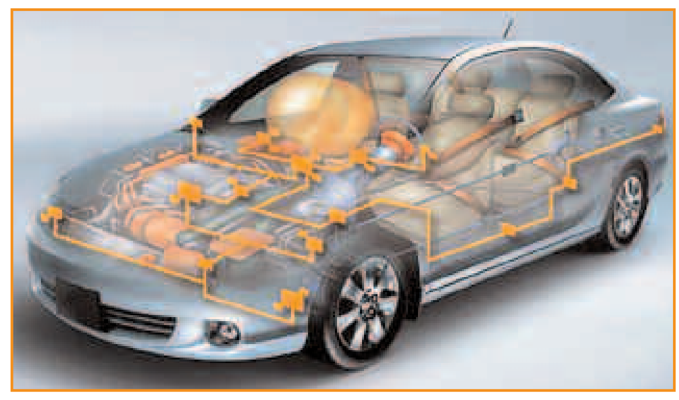
If hardware can be abstracted away, the root of many software problems goes away with it. Programming itself doesn't change, but debugging looks very different. (If debugging is the act of removing bugs, is programming the act of putting them in?) Absolutely any hardware resource (register, memory location, etc.) is visible, testable, and adjustable because nothing's buried inside a chip, board, or box. Data logging becomes trivial and the narrow, bandwidth-limited channel to the emulator, debugger, or trace buffer simply disappears. Breakpoints can be as elaborate as you want because, hey, it's your model.

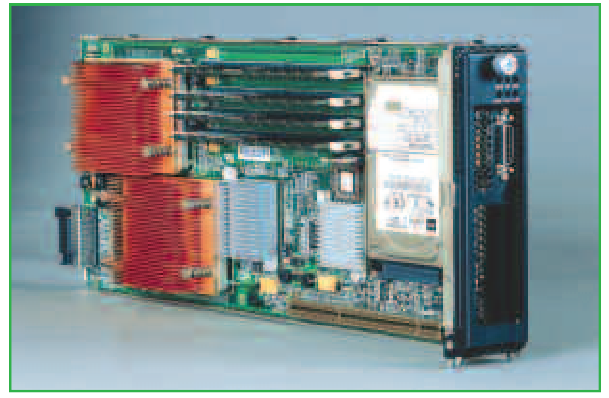
The average developer spends 40% of his time debugging – more time than any other task including programming, writing specifications, or engineering. It's the most time-consuming part of the job, and the part developers hate most. If debugging time can be reduced even a little bit, surely the payoff is worth the investment in model-making. Creating the virtual model is the only new part of the process for most developers; everything else stays the same. But even that's not time lost. The model itself usually serves as the product specification for both the hardware and software teams. It's a "living" executable specification instead of just a paper document. Large companies with benefit from a live working model.

If hardware can be abstracted away, then difficult hardware needn't stand in the critical path. Expensive, complex, or downright impossible hardware systems become a "don't care" for software developers. Multi-core processor chips (and multi-processor systems) are becoming standard fare but how many programmers have experience with them? This is a hardware trend that's not going away, yet most programmers find themselves unprepared for the intricacies that lurk in load balancing, or multi-processor debugging, or task management – to name just a few. How do you set breakpoints in one processor? How does changing the software load on Processor A affect interrupt response in Processor B? And why wait until six weeks before product shipment to find these out? Multiprocessor systems, in particular, cry out for comprehensible and reliable development tools.

There are no hardware bugs if there's no hardware! It may sound glib or simplistic, but this little truism is more practical than one might think. After all, Java programs don't ever crash for hardware reasons, do they? They might be as buggy as any other program but they don't fail because of hardware incompatibilities. If you've thoroughly debugged a new Java program on a PC, it's a safe bet the same program will run on any other Java platform, regardless of how much the hardware may have changed. More to the point, you never had to see the target hardware, or even know anything about it.

What would be nice would be the virtual-hardware aspect of Java but without the run-time overhead on the target system. The large memory requirements and spasmodic performance are okay on a development system but a complete waste of time, space, money, and power on the target system. Oh, and changing languages is still a big hurdle both emotionally and productivity-wise. In summary, what we want is to virtualize the hardware and... that's all.





Jim Turley is an independent technology analyst covering microprocessors, embedded systems, and semiconductor intellectual property. He is the author of seven books, owner and principal analyst of Silicon Insider, the former editor of Embedded Systems Design and Microprocessor Report, and is a regular speaker at industry events. He is frequently quoted in The Wall Street Journal, New York Times, San Jose Mercury News, and appears regularly on television, radio, and Internet broadcasts. Jim lives and works in the Monterey Bay area.

www.JimTurley.com

831.375.8086

sales@vastsystems.com
www.vastsystems.com

North America
Phone: +1 408 328 3300

Europe
Phone: +33 4 56 38 51 23

Japan
VaST Systems Technology KK
Phone: +81 3 6717 2810

South Korea
Daou Xilicon Technology Co. Ltd
Phone: +82 31 789 2800
eda@daouxilicon.com
www.daouxilicon.com

www.virtutech.com

North America
sales_americas@virtutech.com
Phone: +1 408 392 9150

Japan
sales_apac@virtutech.com
Phone: +81 3 6717 6051

Asia Pacific
sales_apac@virtutech.com
Phone: +65 9780 1295

Europe, Middle East, and Africa
sales_emea@virtutech.com
Phone: +46 8 690 0720