

Muscular Methods for Mammoth Designs

By Linda Prowse-Fosler

ABSTRACT

The future of embedded design includes mammoth architectures featuring multi-processor configurations with large numbers of independent busses and bus-bridges. Conventional design and verification strategies are not capable of meeting business and technical objectives at this level of complexity.

This paper first considers significant technical trends in embedded system designs. Conventional embedded design and verification techniques are then reviewed. Finally, the concept of architecture-driven design using virtual system prototyping is presented and discussed.

1. EMBEDDED DESIGN TRENDS

The term “embedded system” refers to an application-specific electronic subsystem that forms a key element in – or sometimes the majority of – a larger system. This main system can range from something as small as a household appliance or a personal communications device to something as large as an automobile, aircraft, or beyond.

A key feature associated with embedded systems is that they feature tightly integrated combinations of hardware and software (referred to as “embedded software”). The software components include system initialization routines, the hardware abstraction layer, a real-time operating system (RTOS) and associated device drivers all the way up to the embedded application code itself

At the time of this writing, the foremost technology drivers in the embedded system space come from the consumer markets for mobile communications (including cell phones and wireless networks), personal digital assistants (PDAs), personal audio and visual entertainment systems, digital cameras, etc.[1] Products such as mobile multimedia computing and communications systems combine the need for massive computational resources with requirements for small size and weight and very low power consumption.

Achieving these goals requires innovative, advanced system architectures that feature multiple processor cores performing different tasks, tiered memory structures with multi-level memory caching, and multi-layer bus structures as well as super-pipelining (running multiple instructions through different pipelines) and super-scaling (adding multiple processors in parallel to achieve higher throughput).

The current state-of-the-art in embedded systems design is to have multiple computational and data processing engines, memory, and peripherals all constructed on a single silicon chip called a system-on-chip (SoC). This paper assumes SoC implementations, although the techniques discussed are also applicable to multi-chip realizations.

1.1 Multiple Processor Cores

By today’s standards, early embedded systems were extremely simple, because they typically used a single general-purpose microprocessor that performed all of the data processing and control tasks. Furthermore, this processor communicated with a

small number of relatively simple memory and peripheral devices by means of a single 4-bit or 8-bit bus.

More recently, it became common for designs to feature multiple general-purpose central processing unit (CPU) cores – coupled with special-purpose digital signal processor (DSP) cores – each with separate caches for instructions and data. In the case of a portable consumer device, for example, one CPU may be used to run the RTOS and service general applications such as messaging and the graphics environment, while another processor might be tightly coupled to a DSP and tasked with handling communications and overall system control.

Having multiple CPUs allows a broader range of processing traffic to be dealt with in real-time, which is a critical requirement for many embedded applications. Today, consumer products typically feature anywhere from two to 10 processing engines, but SoC implementations boasting more than a hundred processors are expected to become reasonably common by the time the 65 nanometer technology node comes online around 2007.

Furthermore, ever-increasing performance requirements – coupled with the demand for real-time responses – are driving embedded designers to take advantage of multiple processors working in parallel. At the same time, the instruction and data widths supported by processing cores has progressed from 4-bit, 8-bit, and 16-bit to 32-bit. Now, the use of 64-bit cores in embedded systems is forecast to grow dramatically[2].

Anytime multiple processor cores are included on a single chip, the tasks of verifying the design and debugging the operation of the entire system become significantly more difficult.

1.2 Application-specific Cores

Early DSPs were relatively generic and could be used to address the needs of multiple applications. By comparison, today’s advanced DSP cores are optimized for very specific applications: a DSP needed by a wireless handset, for example, is very different from one required by a wireless base station. Similarly, the DSP used to process audio data differs from one needed to process digital images, and applications such as modern cell phones may well require both capabilities.

1.3 Multi-level Memory

Another trend in today’s embedded systems is to employ tiered memory structures. Some memory will be tightly coupled to a single processing engine by means of a dedicated bus. Other memory may be local to a group of processing engines while yet more memory may be shared by multiple groups of processing engines. To further complicate the issue, various memory subsystems may have different speed requirements, use different bus widths, and employ a variety of clock speeds and domains.

1.4 Multi-layer bus structures

As was previously noted, early embedded systems often employed a single primary bus to link the processor, memory and peripherals. This bus – which was used to convey both

instructions and data – was essentially a collection of wires connecting all of the devices, where each device could be electrically isolated from the bus by means of appropriate control signals. The operation of such a bus was extremely simple

Today, both CPU and DSP cores can have separate busses for control, instructions, and data, and the various busses feature a wide variety of complex structures and protocols. In addition to general-purpose processor busses, there may be one or more dedicated direct memory access busses along with one or more dedicated peripheral busses. Both the CPUs and DSPs can have tightly-coupled memory busses, external memory busses, and shared memory busses.

Multiple processing engines require mechanisms to communicate between themselves so as to exchange data or control information. Such communication may be via shared memory resources that are accessible from anywhere in the system. (When used appropriately, techniques such as semaphores and mailboxes provide safe passage of information between both processors and peripherals). Alternatively, the inter-core communication may be realized through additional dedicated busses.

Many of these busses will feature pipelined structures (with multiple transaction requests and responses scheduled in the pipeline) coupled with sophisticated cross-point switches that can be attempting multiple read and write operations simultaneously, stacking transactions up, deferring transactions, canceling transactions, and so forth.

The various instruction and data busses are often bridged over to common memory and peripheral busses. Such bridges provide sophisticated links from system-level busses to lower levels in the bus hierarchy. To make things even more complex, the bridged busses may have different widths and may be running on different clock domains at different frequencies.

1.5 Increasing Software Content

The software content of embedded systems is increasing at a phenomenal rate, to the extent that software development and test often dominate the costs, timelines, and risks associated with today's embedded system designs.

For example, each DSP used in a previous-generation product might require around 1,000 lines of C code. By comparison, a DSP used in one of today's applications can easily require 10,000 lines of C code, and this is expected to grow to 100,000 lines of C code over the next few years.

With regards to embedded systems as a whole, a GSM phone circa 2004 may contain approximately 2 million lines of code, which requires an extreme amount of time and resources to develop, integrate, and debug. And the problem is becoming ever more acute, because this number is expected to rise tenfold to 20 million lines of code in a typical cell phone by 2007.

1.6 Decreasing Design Cycles

Coupled with the fact that market windows are continually narrowing, aggressive competition makes today's consumer electronics markets extremely sensitive to time-to-market pressures[3]. With regard to "typical" products circa 2004, design cycle times average 12 to 24 months. However, the market

opportunity for a new product to make an impact can sometimes be as little as two to four months.

The result of failing to have a product available at the beginning of the intended market window may range from significantly reduced revenue (if the product introduction is late) to complete loss of revenue and investment if the window is missed in its entirety.

2. CONVENTIONAL VERIFICATION

In the case of embedded systems, the hardware/software design flow has traditionally been very sequential in nature. That is, the overall architecture of the system is determined; the hardware is designed and a hardware prototype is constructed; the operating system and/or middleware is installed and tested; then the embedded software is developed, ported, integrated, and debugged.

A key impact of the shorter design cycles noted in the previous section is that it is no longer economically viable to wait for an SoC to be fabricated before starting to develop the software, integrate the software and hardware, and verify and debug the system. Similarly, it isn't viable to wait for the implementation level representation of the design (specified in register-transfer language (RTL) using a hardware description language (HDL) such as Verilog or VHDL) to become available before fully committing resources to developing the embedded software.

2.1 Physical Prototypes

For an embedded system design, a physical prototype typically involves a circuit board and the SoC in the form of working silicon. However, this means that the hardware portion of the design is now almost 100 percent tied down (with the exception of any field-programmable fabric incorporated into the SoC), which makes this approach essentially useless in the context of exploring and evaluating alternative architectures.

The point is that, in the case of today's extremely complex embedded systems, important hardware/software tradeoffs must be made long before the design partitioning is locked down and the chips are manufactured. Relying on a physical prototype as the primary verification and debug mechanism means that these tradeoffs must be largely based on experience and intuition, as opposed to having any substantial amounts of hard data available. This may have been acceptable in the case of simple single-processor designs, but it is not feasible in the context of today's mega-complex algorithms running on multi-core systems with tiered memory and multi-layered bus structures. The lack of appropriate architectural evaluation means that the system may well fail to achieve the required levels of bandwidth and performance. In turn, this may result in products such as cell phones failing to achieve the required quality of service.

The sheer size and complexity of today's embedded system designs means that – when relying on a physical prototype-based approach – it is almost inevitable that a substantial number of bugs will end up in the silicon. A physical prototype provides only limited visibility into the internal behavior of the fabricated chip, which makes hardware/software debugging at this stage an extremely daunting task. For example, when it comes to debugging failing tests, it is extremely difficult to coordinate and

synchronize hardware verification tests involving multiple processors.

Even after all of the bugs have been detected, isolated, and resolved, re-spinning an SoC is prohibitively expensive and – worse – will often cause the product to miss its ideal market window. And even if – by some miracle – no bugs did make it into the hardware, delaying the commencement of the bulk of the software development until the physical prototype becomes available means that, once again, the product is likely to miss its market window.

Last, but certainly not least, the cost of designing and fabricating a physical prototype is expensive to begin with and, in the case of products requiring large numbers (sometimes hundreds) of software developers to have access to the development platform, there are significant costs associated with creating multiple copies. This cost increases when coupled with the logistical problems involved in shipping the prototypes around the world and getting them operational in their various locations.

2.2 Hardware Acceleration and Emulation

Hardware-based simulation acceleration and emulation solutions typically involve arrays of field-programmable gate arrays (FPGAs) or processors. These solutions accept RTL representations of the design and translate them into an equivalent suitable for hardware acceleration.

There are several problems associated with these solutions, not least that they can be very expensive and each unit can be used only by one (or very few) software developers at a time. There are also issues in the context of designs featuring multi-processor environments, because individual emulator tools often lack the appropriate synchronization mechanisms to be able to work in debug mode in a multi-processor environment.

Such emulators also have problems with limited visibility into the design (although this is not as bad as with the physical prototypes discussed above). However, the real show-stopper with regard to these verification and debug solutions is that they require RTL representations of the hardware portions of the design, which means that software development cannot commence in earnest until a long way into the design cycle. In turn, this means that the hardware portion of the design is largely established, which once again limits this approach with regard to exploring and evaluating alternative architectures.

2.3 RTL-based Simulation

As for hardware-based acceleration emulation, an RTL simulation solution requires RTL representations of the hardware portions of the design, thereby delaying meaningful software development until the RTL becomes available.

Trying to determine how well the design's architecture performs in the context of real software workloads simply isn't possible in the context of a software simulation. A software simulation running on a on really high-end (and correspondingly expensive) machine would be lucky to achieve equivalent simulation speeds of more than a few Hz (that is, a few cycles of embedded system clock for each second in real time). Practically, this means that detailed simulations can be performed on only small portions of the software.

One solution that goes some way toward addressing these issues is to use a simulation farm involving a network of computers, each performing a portion of the simulation. However, this can be a tremendously expensive solution, especially if the environment needs to be replicated in multiple facilities that may well be located in different countries around the world. Furthermore, in the case of simulating an embedded design featuring multiple cores, busses, memory elements, and peripherals, there is typically a huge amount of inter-computer communication. The result is often that numbers of the simulation engines spend significant amounts of time waiting for a portion of the design located in another machine to finish its processing before they can access the results and make use of them.

2.4 ISS-based Simulation

The traditional alternative to all of the previously described verification approaches is to attempt to verify and debug the chip using software-based models[4]. The most common approach is to use an instruction set simulator (ISS) for each processing core in the design. These models can execute the same binary code as the actual processors, giving them the potential to be used to develop and debug software prior to SoC fabrication. However, only processor cores are modeled using ISS representations, while the remaining design units (logic, memory, peripherals, etc.) still have to be coded at the RTL level (or sometimes as C models).

It is generally accepted that any form of simulation running at an equivalent instruction processing rate of less than 10 MIPS cannot satisfy software engineers' requirements when it comes to performing edit-compile-run-debug cycles. Traditional ISS solutions typically achieve only a few MIPS, which makes them far too slow to run a large body of software such as an RTOS, and thus renders these solutions capable of verifying only small portions of the embedded code. By comparison, in the cases of ISS-based solutions that can achieve higher performance, this is invariably realized at the expense of accuracy.

The majority of ISS models are classed as providing only instruction-level accuracy. This means that, when running code, they will faithfully reproduce the same results as the processor core in the real chip. However, their cycle-by cycle behavior may not exactly match the physical core's interfaces. Similarly, an ISS-based solution will typically not match the operation of multi-layer bus structures in a cycle-accurate manner. Due to the fact that accesses to caches and memory are not cycle accurate, memory performance cannot be measured precisely, thereby limiting the user's ability to make detailed architectural tradeoffs.

With regard to debugging, the use of multiple discrete ISS models imposes significant limitations. The lack of synchronization mechanisms between the individual models means that it is difficult to understand how multiple processors are interacting when trying to debug a failing test. Also, as opposed to occurring in the cores themselves, many errors occur due to timing issues in the environment surrounding the cores (such as race conditions and deadlocks), and these problems cannot be detected by single-stepping.

3. VIRTUAL SYSTEM PROTOTYPES

The term virtual system prototype refers to a software simulation-based, cycle-, register-, and timing-accurate electronic systems

level (ESL) model. Such a model can be used to represent one or more portions of an embedded system or the entire system. Unlike first- and second-generation ISS-based solutions, today's virtual system prototype environments, which are available from companies such as VaST Systems Technology (www.VaSTsystems.com) and Virtio Corporation (www.virtio.com) are based on state-of-the-art third-generation software architectures.

One key feature of virtual system prototypes is that they offer the ideal combination of performance and accuracy[5]. A single-core simulation can achieve 50 to 200 MIPS, while multi-core systems with tiered memory structures, multi-level buses, bus-bridges, and peripherals can be simulated at ten to 100 MIPS depending on the configuration. This is sufficiently fast to support real-world tasks, such as booting an RTOS in a matter of seconds. In addition to speed, the cycle- and timing-accurate models forming a virtual system prototype can run the same compiled and linked target code as the real machine while accurately representing the systems real-world behavior. Meanwhile, the integrated environment provides real-world synchronization between all of the cores, busses, memories, and peripherals.

The advantages of using a virtual system prototype of an embedded system design are multifold. They include the facilitation of architectural exploration and evaluation which – when coupled with the virtual system prototype ability to act as an executable specification and golden reference model – lead to the concept of architecture-driven design. Furthermore, a virtual system prototype provides the ability to concurrently develop the software and hardware portions of the design. In addition to cutting an average of nine months from a typical embedded system design, this minimizes the occurrence of downstream bugs, dramatically reduces the occurrence of silicon re-spins, and provides a high level of confidence in the final deployed product.

3.1 Architectural Exploration

Virtual system prototypes facilitate architectural exploration and evaluation by allowing hardware and software components to be mixed and matched. Accurate measurements that model real-world behavior under real-world data processing and software workloads allow system architects to make accurate hardware-software tradeoffs early in the development process.

Typical issues relate to bus bandwidths and the latency impacts associated with heavy traffic. Consider the process of encoding and decoding video data, for example: such coding and decoding can be performed in hardware or software. The latter may be the preferred solution for a particular application, but the issue is ensuring that there is sufficient processing bandwidth in a software implementation before building the system. Such issues are of particular concern for real-time applications such as speech analysis and synthesis, in which performance guarantees absolutely need to be met. In an attempt to ensure that performance goals are realized, a common scenario in embedded system designs is to over-engineer the product. This costs time and money, but the alternative – a system that ends up with insufficient processing power – can have huge costs associated with the project being delayed due to design re-spins.

Virtual system prototypes can be used to better understand hardware and software partitioning decisions and to evaluate the

throughput considerations associated with alternative implementations. The ability of a virtual system prototype to run real software workloads allows system architects to analyze performance considerations and impacts, such as cache sizing, processor capacity, and bus bandwidth issues, and also to detect resource sharing contentions and potential synchronization problems.

3.2 Architecture-level Power Analysis

Accurate power analysis should ideally occur throughout the design, commencing with the architectural evaluations and undergoing subsequent refinement as the design progresses to a final implementation. In the case of conventional design and verification environments, power modeling in the early (architectural) stages of a design is typically performed using experience, intuition, and “guestimates,” all gathered together in the form of a spreadsheet. By comparison, in the case of a virtual system prototype, each subsystem can include a calculator function to model power consumption. The virtual system prototype can monitor the exact number of data transfers over time, and can correlate such data throughout the system

3.3 Third-party IP Evaluation

One aspect of using a virtual system prototype that may not be immediately apparent is its use in the role of third-party intellectual property (IP) evaluation. When a design team is presented with a number of IP alternatives, it can be extremely difficult to make comprehensive evaluations based on datasheets and other “paper” specifications. Raw computational and “throughput” values can be misleading, because real-world performance can be strongly affected by the type of data being processed and the way in which IP blocks are integrated into the rest of the system.

However, if each IP core under evaluation comes equipped with an equivalent virtual system prototype model that can be incorporated into the full system prototype, the comparison of different solutions in the context of the full system environment with real software loads provides accurate and meaningful results upon which to base design decisions.

3.4 Executable Specifications

The inherent complexity of multi-core embedded systems with tiered memory and multi-level bus structures is rapidly becoming unmanageable with regard to “paper” specifications captured in natural language. In many cases, such specifications are open to misinterpretation or multiple interpretations, and the likelihood of errors creeping into the design during the course of the implementation process is high.

This is driving a requirement for “executable specifications” against which hardware and software engineers can compare their implementations so as to ensure consistency throughout the design flow. This is one of the features provided by a virtual system prototype, which is functionally (cycle- and register-) accurate compared to the downstream RTL and final silicon. Thus, once the initial architectural model has been built, it becomes the executable system specification that drives the concurrent development of the detailed hardware and software implementations.

3.5 Concurrent Hardware-Software Design

One of the biggest advantages of a pure software simulation solution such as a virtual system prototype is that it can be made available nine or more months earlier than physical hardware. This allows embedded designers to start developing software long before the actual silicon and the rest of the hardware portions of the system are ready[6].

In the case of a wireless chipset manufacturer, for example, a big issue is that hardware providers' customers require early representations of the mobile chipset and base station chipset so that software development can begin as soon as possible. The development times and product life cycles in this marketplace are now so short that any delays in hardware availability that cause software development to be pushed out are potentially catastrophic. In this type of market, hundreds (sometimes thousands) of software developers need to start working on everything as early in the design process as possible, from the system initialization routines and the hardware abstraction layer through the RTOS and associated device drivers all the way up to embedded application code itself.

The fact that a virtual system prototype can be delivered well in advance of actual hardware – and also that it can be made available almost instantaneously through the internet to global engineering resources located in facilities worldwide – promotes the efficiency of a globally distributed engineering workforce and software development base. Furthermore, the fact that new versions of a virtual system prototype can be made available to all of the software and hardware engineers practically simultaneously helps to keep the development process synchronized.

A major benefit of this concurrent hardware and software development is that analysis of the way in which real software workloads perform may influence the hardware design early in the process before anything is committed to silicon. The end result can be more efficient, higher performance products.

3.6 Visibility into the Design

As was previously noted, a major problem associated with physical prototypes and hardware emulation solutions is that they provide only limited visibility into the internal behavior of the design, which makes hardware-software debugging an extremely daunting task. By comparison, a software virtual prototype provides virtually unlimited real-time access to internal probe points throughout the design. The use of techniques such as associating protocol checkers with busses and interfaces facilitates the detailed concurrent debugging of the hardware and software in the target system.

Another big problem with embedded system designs is that the software can have a lot of subtle dependencies on low level aspects of the hardware implementation. For example, it may be necessary to initialize a set of registers in a specific order; if this initialization is not performed correctly, the result can be subtle problems that can be difficult and time-consuming to track down. It is often the case that this sort of situation is described and documented early in the design process, but it is easy for this information to become lost or overlooked, especially when hundreds of software developers are involved, new members join the team, entire new teams come online, and so forth. The answer is to embed checks that monitor these low level dependencies

directly into the software virtual prototype. This means that if a software routine initializes something incorrectly or in the wrong order, the developers receive immediate and explicit warnings that can save hours, days, or even weeks of downstream debugging effort.

3.7 Multi-core Debugging

The complex interactions associated with designs featuring multiple cores and peripherals mean that there are many communications links that require deep observation and debugging. For example, in the case of a system relying on synchronization through memory, common pitfalls include improper arbitration of memory accesses and data corruption caused by one subsystem modifying data before another has accessed it. An alternative approach is to use direct communication through dedicated host ports, but systems employing this method are prone to synchronization problems that can cause stall conditions and deadlocks, both of which can be challenging to debug.

Key verification and debugging issues that need to be addressed include shared memory applications (race conditions and data corruption), direct communications links (deadlocks, stalls, and starvation), processor performance (cache usage, pipeline stalls, and starvation), bus performance (bandwidth and congestion), and peripheral performance (latency).

Virtual system prototypes make it easier to debug all of these problems by combining full visibility into the design with a comprehensive environment fast enough to execute actual code and accurate enough to replicate low-level hardware-software interactions.

3.8 Regression Testing

With regard to today's incredibly complex software systems, a small change in one portion of the code can easily destabilize other portions of the design, often in subtle ways. Detecting and isolating these problems can be difficult, especially when large numbers of developers are working on different aspects of the code. Thus, in the traditional software world, it is common to run regression suites every evening so as to quickly identify and address problems as they occur.

This type of regression testing is very difficult to achieve with traditional embedded systems verification solutions. This is especially true in the case of physical prototypes, which require a lot of "hand-holding" and whose outputs may well be in terms of logic analyzer signals. In these cases, it can be very labor intensive to download tasks into the hardware, and to then run those tasks and collate and analyze the results.

By comparison, scripting techniques can be used to load tests into a virtual system prototype, run those tests, and capture and check the results. Furthermore, virtual system prototypes provide the ability to monitor and/or stimulate internal nodes that are not available in the actual product, thereby allowing for a more detailed and higher quality verification of the system than can be achieved with the actual physical product. Such scripts can be used to automatically run suites of regression tests every night, which results in dramatic savings in the time it takes to detect, debug, and resolve problems as they are introduced into the code.

These capabilities also enhance software developers' abilities to introduce deliberate faults to see how a system responds. This allows the developers to test system recovery situations and understand the ramifications of a variety of potential problem scenarios, such as communication path failures, RAM and ROM corruption and recovery, etc.

3.9 Benefits to Hardware Design Engineers

The major benefits associated with software virtual prototypes with regard to hardware design engineers are less wasted RTL code development and higher quality products. Using a software virtual prototype for architectural exploration and evaluation with real software workloads helps ensure that the architectural specification handed over to the RTL design engineers is not subject to radical changes in the downstream portions of the development cycle.

Furthermore, the fact that the virtual system prototype is cycle- and register-accurate means that the prototype as a whole – and the various subsystem models forming it – can be used in the role of “golden reference models” against which the RTL implementations can be automatically simulated and compared to ensure that the RTL accurately reflects the design's functional intent.

4. CONCLUSIONS

At the time of this writing, it would not be considered unusual for an embedded system to contain six-plus processor cores, 40-plus busses, and 30-plus bus-bridges. This trend is set to continue, and embedded systems containing more than a hundred processors are expected to become reasonably common by the time the 65 nanometer technology node comes online around 2007

The rapidly increasing software content of embedded products, coupled with shrinking development times and shorter product life cycles, makes it mandatory to commence software development concurrently with hardware development.

Existing embedded system verification strategies – such as physical prototypes – require the hardware to become available before the main software effort can begin. Other approaches require access to RTL representations of the design, which only become available late in the design cycle, and which can typically be used to verify only small portions of the code.

By comparison, virtual system prototypes based on third-generation software architectures offer the ideal combination of performance and accuracy. These software prototypes can run unmodified binary executables from boot sequences to operating systems to embedded applications; they facilitate sophisticated architectural exploration and evaluation; they allow hardware and software development to take place concurrently; and they provide executable specifications and golden reference models.

The fundamental end result of using a concurrent hardware-software design approach is a reduction in the total engineering time required to develop a product ready for production. Overall, the use of virtual system prototypes can result in the development of better products with shorter development times for less money.

5. REFERENCES

- [1] Summary presentation at the *Fall Processor Forum*, San Jose, CA, 2004.
- [2] Report on *Embedded 64-bit Microprocessors in Customer-Specific, Cell-Based Designs: Upping the Bandwidth*, Instat/MDR (www.instat.com).
- [3] Ahmed A. Jerraya, *Long Term Trends for Embedded System Design*, TIMA Laboratory.
- [4] Z. Vojin, C. Schlager, J. Fitzner, *System-level Modeling of DSP and Embedded Processors*, AXYS Design Systems (www.axys.com).
- [5] G. Ahn (VaST Systems), P. Rao (StarCore LLC), *System-level Debugging in a Multi-Core Wireless Virtual System Prototype* (www.VaSTsystems.com).
- [6] F. Winters (Delphi Corp.), C. Mielenz (Infineon Technologies AG), G. Hellestrand (VaST Systems), *Design Process Changes Enabling Rapid Development*, Convergence Conference, 2004