Learning From Advanced Hardware Verification for Hardware Dependent Software

Simond Davidmann, Duncan Graham
Imperas Software, Ltd
www.imperas.com

Abstract— We present a new perspective for embedded software verification for generalized multicore processor platforms, somewhat analogous to simulation-centric hardware verification solutions. A spatial, temporal, and abstract multi-dimensional framework for software verification, profiling, analysis, and debug is proposed that leverages a specialized simulation core. The simulator enables key services for the verification solution while providing a degree of separation from both the hardware models and software under test, to ensure accurate behavioral representation as well as customization and performance advantages. We include two real world examples of the use of this framework.

Keywords—software, hardware, debug, verification, analysis, profiling, simulator, simulation, multicore, processor, OVP

I. Introduction

Modern multicore platforms present a range of new challenges to embedded software developers. The most significant of these occur at the hardware software boundary, namely the Operating System, Drivers and other code that interacts directly with the hardware. We term this "Hardware Dependent Software" or HDS.

HDS code verification requires a range of analysis and debug solutions that combines techniques from classic software development with methods utilized for hardware test. A verification perspective that encompasses the complexity introduced by multicore processor device sharing, intricate operating system execution, and the nature of generalist hardware platforms running specific software applications is key for a rigorous test of the entire system.

II. Embedded Platforms and Software Architectural Trends

Standardized multicore processor platforms, with embedded software utilized to drive specific application functionality, are now commonplace in modern electronic devices. This has resulted in a dramatic increase in the amount of embedded software in a device, and its complexity. Platforms often include multicore processors, signal processors, accelerators, memory configurations, security infrastructure and a broad range of peripherals [1].

Many modern platforms employ a "Symmetric Multi-Processing" (SMP) multiprocessing architecture [2]. In this configuration, near identical processing cores communicate through shared memory, and a single operating system, such as Linux, will schedule tasks across the cores. Platforms also leverage Asymmetric Multi-Processing (AMP) architectures, where the processor cores are typically different and usually dedicated to specific jobs, for example, a DSP executing dedicated algorithms.

All of these architectures have a significant impact on the embedded software stack, particularly that code that interacts directly with the hardware.

The software solutions that drive the functionality of these devices can be equally complex. For some platforms a "bare-metal" software configuration with no OS or a Real-Time Operating System (RTOS) supporting a single application is sufficient. However, more elaborate software architectures that are based on the Linux Operating System (OS) are become more common.

These solutions may include a large number of drivers, hypervisors to support virtualization, middleware libraries to provide application functions, and a range of applications that leverage a comprehensive common user interface. Figure 1 shows an example Android-based software stack.

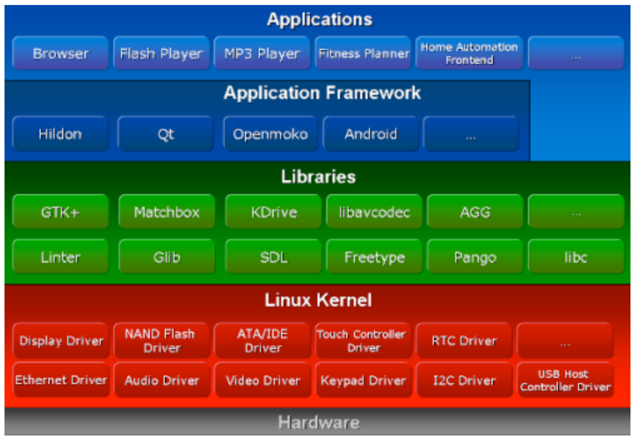


Fig 1: An Android (Linux) software stack example (courtesy Promate, Inc.).

Obvious examples of HDS are operating systems and drivers, but can also include hypervisors that make use of a platform's

Memory Management Unit (MMU) to create virtual compute environments, or specific middleware code that works in conjunction with hardware accelerators or DSPs.

In general, HDS code has specific development requirements that relate to its interaction with the hardware. The verification of HDS software is inevitably more rigorous than general applications as it is often harder to change post-production and has a greater influence on the entire system. Problems in this code are also often harder to find, as their effects can manifest themselves elsewhere in the system. Issues created by unusual, corner case scenarios involving both hardware and software states can occur more easily. The performance of HDS code and its interaction with hardware components can have a greater impact on the entire system.

III. TRADITIONAL EMBEDDED SOFTWARE DEVELOPMENT SOLUTIONS

Embedded software verification has traditionally leveraged hardware prototypes, using unwieldy interfaces, such as JTAG, to extract signal information. This limits signal visibility and system controllability. In addition, software test must wait for a prototype to become available. Emulation and FPGA-based hardware prototypes mitigate these issues, providing greater visibility and controllability, and earlier availability. However, these are often expensive alternatives, low in number and so availability, do not have the performance of final hardware, and require additional engineering.

Timed, cycle-based models, often derived from the Hardware Description Language (HDL) code used to create the final device, have allowed earlier software verification. The issue with these models becomes one of trading off required hardware functionality for software performance [3]. As large portions of the hardware operation are reproduced in the model, the greatest performance that can be derived is of the order to 10,000 times slower than final hardware, an impractical solution for test at all levels of abstraction. Occasional cycle accurate simulation can be useful for specific issues.

For software verification, it has proven unnecessary to mimic either the full timing or operation of the platform. An "Instruction Accurate" (IA) model eliminates much of the data processing and abstracts the timing to instruction execution ordering. With these models it is possible to achieve software execution rates in the real time range of 100s of MIPS, or above, while providing the required level of functionality.

A Virtual Platform is an IA software representation of a processor plus other key platform components required to test an embedded software stack. Many modern Virtual Platforms make use of simulation technology and enable the use of advanced debug and verification tools.

Various standards have evolved to make Virtual Platform modeling somewhat easier, for example:

1) SystemC [4]: an open source set of classes and macros that provide a simulation kernel with a C++ API to which models may be coded.
2) The "Quick EMUlator" (Qemu) [5] is an open source hypervisor, which provides an emulated API for a host machine to which models may be coded.
3) Open Virtual Platforms (OVP) [6] is a high-performance simulation solution with C APIs, to which models may be coded.

All of these solutions provide simulation style services eliminating a lot of programming for the modeler. In this paper we will base our discussion on the OVP solution.

HDS code exists on the boundary between software and hardware development. As such it requires capability from both domains. Furthermore, the more stringent requirements demand enhanced functionality. These debug and verification requirements may be subdivided by considering the multi-dimensional nature of the tool suites. We would refer to single dimensional tooling as a relatively standard examination of serial software execution, two dimensional tooling that incorporates the notion of both spatial (looking across the multicore platform) and temporal (deeper event timing consideration), and three dimensional tooling where the software stack abstraction is leveraged to improve insight.

IV. SIMULATION-BASED VIRTUAL PLATFORMS

The notion of using simulation in the software engineering space is unusual. However, when using a virtual platform to model a real world system, simulation offers valuable benefits not dissimilar to those for hardware verification.

The OVP virtual platform architecture, see fig 2, leverages the simulator as the central component to enable a high-performance reproduction of the platform. The simulator takes care of the mechanics of the platform model operation, including simulation operation, scheduling, and access for tool controllability and visibility.

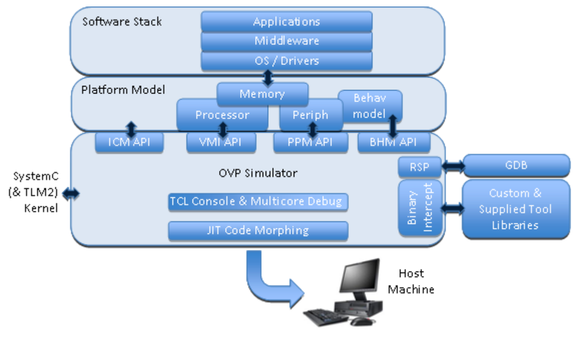
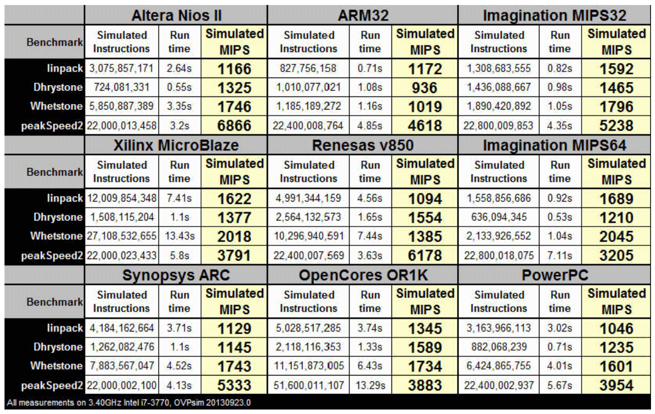


Fig. 2. OVP Virtual Platform Simulation Interfaces.

The simulator provides APIs for the modeling of key processor and platform functions, without concern for detailed simulation operation, for example the Open Virtual Platforms (OVP) APIs [7].

A well-architected simulator will accelerate the performance of a virtual platform model significantly. As has been shown in the hardware domain, simulation performance level is directly proportional to the quantity of tests that may be applied, and therefore vital for rigorous verification. Fig 3 shows some of the performance levels that may be achieved using the Imperas OVP simulator, as an example. In some cases the real time performance of the actual device is exceeded.



	Altera Nios II			ARM32			Imagination MIPS32		
Benchmark	Simulated Instructions	Run time	Simulated MIPS	Simulated Instructions	Run time	Simulated MIPS	Simulated Instructions	Run time	Simulated MIPS
linpack	3,075,857,171	2.64s	1166	827,756,158	0.71s	1172	1,308,683,555	0.82s	1592
Dhrystone	724,081,331	0.55s	1325	1,010,077,021	1.08s	936	1,436,088,667	0.98s	1465
Whetstone	5,850,887,389	3.35s	1746	1,185,189,272	1.16s	1019	1,890,420,892	1.05s	1796
peakSpeed2	22,000,013,458	3.2s	6866	22,400,008,764	4.85s	4618	22,800,009,853	4.35s	5238
	Xilinx MicroBlaze			Renesas v850			Imagination MIPS64		
Benchmark	Simulated Instructions	Run time	Simulated MIPS	Simulated Instructions	Run time	Simulated MIPS	Simulated Instructions	Run time	Simulated MIPS
linpack	12,009,854,348	7.41s	1622	4,991,344,159	4.56s	1094	1,558,856,686	0.92s	1689
Dhrystone	1,508,115,204	1.1s	1377	2,564,132,573	1.65s	1554	636,094,345	0.53s	1210
Whetstone	27,108,532,655	13.43s	2018	10,296,940,591	7.44s	1385	2,133,926,552	1.04s	2045
peakSpeed2	22,000,023,433	5.8s	3791	22,400,007,569	3.63s	6178	22,800,018,075	7.11s	3205
	Synopsys ARC			OpenCores OR1K			PowerPC		
Benchmark	Simulated Instructions	Run time	Simulated MIPS	Simulated Instructions	Run time	Simulated MIPS	Simulated Instructions	Run time	Simulated MIPS
linpack	4,184,162,664	3.71s	1129	5,028,517,285	3.74s	1345	3,163,966,113	3.02s	1046
Dhrystone	1,262,082,476	1.1s	1145	2,118,116,353	1.33s	1589	882,068,239	0.71s	1235
Whetstone	7,883,567,047	4.52s	1743	11,151,873,005	6.43s	1734	6,424,865,755	4.01s	1601
peakSpeed2	22,000,002,100	4.13s	5333	51,600,011,107	13.29s	3883	22,400,002,937	5.67s	3954

All measurements on 3.40GHz Intel i7-3770, OVPsim 20130923.0

Fig. 3. OVP Simulation Performance for Various Processor Types.

Another advantage of simulation is the access that may be provided for debug and verification. The simulator allows many aspects of a platform execution to be observed, from hardware components through to abstract software data structures. The simulator is also more flexible in terms of controllability, for example, changing code dynamically, stopping and starting execution, etc.

A simulator allows some decoupling between a tool suite and the code on which it is being applied. This can be extremely useful to preserve execution operation and eliminates the execution altering behavior present in some other solutions, so called “Heisenbugs” [8].

For multicore processor platforms, simulation-based solutions provide further advantages, in the area of increased timing and support for analysis and debug that operates across all cores rather than focused on one.

V. Single and Two Dimensional, Multicore Debug and Analysis

It is commonplace now for software debug and analysis solution to have a full range of single dimensional capabilities. These include debug features such as tracing and visualizing a range of objects, and execution control such as breakpoints. Analysis features include code profiling for performance bottlenecks, static code analysis, and code coverage. Of more interest is the debug and analysis of multicore HDS on which this section is focused.

AMP cores tend to operate independently from one another, so platform level verification in a pure AMP environment is generally focused on control operations between a master processor and other cores. As such multicore verification is limited to the occasional control operation.

SMP environments where multiple cores work together, sharing memory and running under a single Operating System, do require additional, cross platform verification capability. In addition, an element of time to varying levels of granularity is also required. We would define this level of capability as 2-dimensional debug, in the spatial and temporal domains.

One requirement of any analysis and debug tool suite in an SMP environment is the ability for it to operate on and examine all the cores concurrently and to allow tasks scheduled by the common OS to be examined together. Indeed on some occasions it is useful to be able to perform fully cross platform operations, checking variables or events at various points. More typically an engineer is focused on one part of a platform and/or area in the code, but needs to incorporate some detail from another part of the platform or code base. This may be accomplished with introspection.

Introspection is generally defined as the ability of a program to examine the internals of an object during runtime [10]. In our context, an introspective tool brings together information if available and presents or uses it for a specific operation. On a multicore platform introspection is a critical component as it allows complex tool configurations to be created, together with a degree of automation.

For example, imagine a communications device driver sending an interrupt to a processor, to activate an interrupt service routine to handle incoming data. An analysis breakpoint could be set on the call of the service routine, which in turn calls back to a system analysis function. This function then introspects the system, allowing a level of reusability for the analysis function to check available objects on any task.

Temporal analysis and debug is the examination of system execution over a period of time. Software engineers generally consider time at a higher level of granularity than that required for HDS analysis. The scheduling of tasks and/or threads across the cores, and their interaction, can lead to issues both in terms of their operation or, more commonly, avoidable performance bottlenecks. This requires a more structured time visualization approach that may include an indication of real time. In this case the timing is structured such that it does not cause the entire simulation to slow down.

The obvious tool for this work is a waveform or timing chart type of display, see fig 4, a staple of the hardware debug world, where a database of events is recorded and displayed over time. This has indeed been implemented in a number of environments mainly for the analysis of concurrent operations. For the purposes of HDS where some timing information is required, more advanced timing display analysis such as cause and affect tools are also being used.

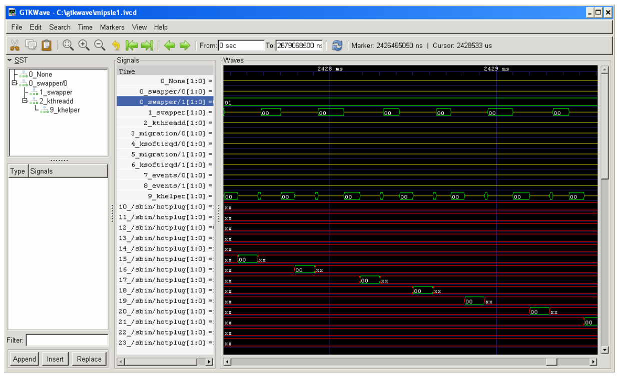


Fig. 4. OS Schedule Analysis

Consider the previously described interrupt service routine situation where the OS suspends tasks to service the interrupt, with an associated control procedure that might involve semaphores, the stack, etc. The serial operation of classic software debuggers is inadequate to catch possible problems. A temporal based solution shows the sequence together with the status of the other variables in a 2 dimensional view making possible multicore generated conflicts clearly visible.

Many time-based checks can be improved by the use of temporal assertions. The use of standard C assertions has proven inadequate for anything but the most simple of checks. Using the simulation temporal callback mechanism it is relatively easy to construct an introspective state machine style assertion that may also make use of other platform data. As with other analysis structures using the simulator, these checks are efficient and separate from the source code.

VI. THREE DIMENSIONAL LAYER AWARE ANALYSIS

Embedded software is normally architected in terms of layers, giving rise to the notion of a "Software Stack" This might be likened to the hardware abstractions now being utilized in standards such as Accellera's Unified Verification Methodology (UVM) [11]. So if the software itself is designed in layers to increase comprehension, then it follows that the verification process may also benefit from the same approach. This is the notion behind 3-dimensional debug and analysis; a suite of tools providing capability that operates in the spatial, temporal and, now, abstract dimensions.

Two modes of layered operation are required: The examination of activity through the layers, for example checking OS events together with CPU detail, and the ability to focus on activity specific to a layer, while eliminating irrelevant detail. An example of this second mode is "OS Aware" analysis.

The actual requirement for this verification mode is slightly more granular than the general definition of a "layer." The sub-layers that make sense for HDS verification are:

- Operating System
- Bare Metal Apps & some Middleware
- Platform code (e.g. Drivers)
- Processor

Each one of these sub-layers can make use of the commands available, but often also benefits from specific commands. For example, specific "OS Aware" commands, commands that are based on OS features, are extremely useful when porting an OS, such as Linux, to a new platform.

As an example, the Imperas OS Aware profiling tools allow the use of the following example operations for Linux, as well as other operating systems:

- Tracing scheduled events, console output, execve calls, etc.
- Console window manipulation
- Task status inspection
- Various control and break on OS events

When used in conjunction with other platform and cpu operations, for example:

- Shared memory and cache analysis
- Peripheral inspection
- Virtual memory mapping
- Calls to a hypervisor or secure monitor
- Break on exception

some powerful analysis capabilities are possible. To bring up an OS on a new platform, an understanding of OS operation for that particular platform is essential and these commands fulfill this need. Figure 5 shows where the operations fit in a layered verification hierarchy.

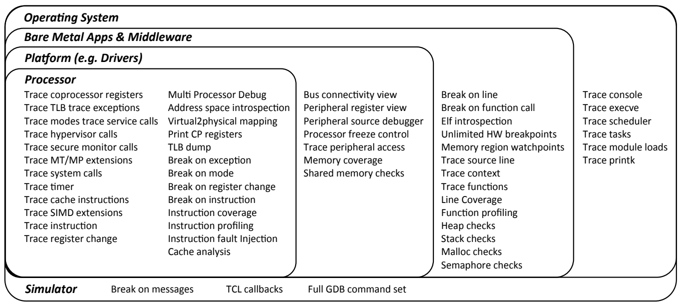


Fig. 5. Layered Operations

In addition to the general-purpose layers, it makes sense to include a library of capabilities that provides commands to specific CPUs or Operating Systems. The libraries may also be augmented with commands pertaining to a particular platform configuration, including the application and library software and the custom APIs it may use. It is this level of customization that allows a platform team to create a library of capabilities to go with their specific product. Figure 6 shows a diagram of the OVP tool library layering architecture, designed with this in mind.

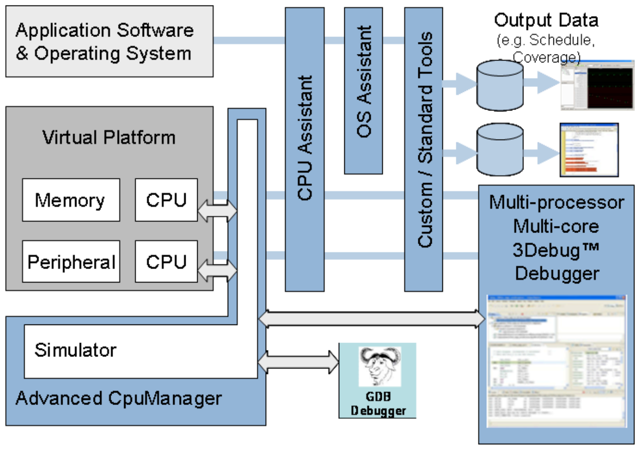


Fig. 6. Layered Tool Architecture

For example, the Imperas Linux tool library includes commands to trace a range of OS activity specific to Linux, including scheduler operation, execve calls, printk output, console output, tasks and module loads. It also allows the status of tasks running to be checked, to break and load the symbols on a load of a kernel module (note this uses introspection to obtain the addresses where the symbols are located in the virtual memory space), and other Linux specific operations.

For example, the "Schedulertrace" command utilized in the OVP environment to trace and display task activity is used on any CPUs and OSs using the relevant customization layers. Figure 7 shows the command issued from a Makefile for specific use with Linux running on the MIPS Malta platform, which is configured for this specific simulation.

```
schedulertrace::$(MIPS_MALTA_VMLINUX) $(MIPS_MALTA_INITRD)
imperas.exe --finishtime 3.0 \
--vlnvname MipsMaltaLinux --vlnvvendor mips.ovpworld.org \
--override MipsMaltaLinux/mipsle1/variant=34Kc \
--override MipsMaltaLinux/mipsle1/imagefile=$(MIPS_MALTA_VMLINUX) \
--override MipsMaltaLinux/Core_Board_SDRAM_promInit/initrd=$(MIPS_MALTA_INITRD) \
--override MipsMaltaLinux/Core_Board_SDRAM_promInit/kernel=$(MIPS_MALTA_VMLINUX) \
--override MipsMaltaLinux/mipsle1/enableSMPTools=1 \
--enabletools \
--extlib MipsMaltaLinux/mipsle1=linuxOsHelper \
--callcommand "MipsMaltaLinux/mipsle1_TC0/vapTools/schedulertrace -on"
```

Fig.7. Schedulertrace Makefile for the MIPS Malta Platform running Linux

This Makefile target loads a model of the MIPS Malta Platform available on the OVP website [12] and starts a simulation. The "callcommand" argument allows the scheduler trace tool to be started from the command line and the "finishtime" option stops the simulation after 3.0 seconds. The Scheduletrace output generated is shown in Figure 4 above, and displays the scheduled tasks across all processor cores over time.

The layered approach may be augmented with a custom tool overlay that allows standard verification techniques to leverage OS and CPU aware commands, where it makes sense. This enables verification commands, such as coverage, fault injection, and memory profiling, to be applied to specific platforms and OSs using the simulation infrastructure.

VII. CUSTOM TOOL CAPABILITIES

It is common in software development, more than in hardware, to add test lines into the actual embedded code, although such code can affect software behavior and system performance. One of the most useful aspects of simulation-based software verification is the ability to add custom tool code to the simulation API that operates unobtrusively from the software and models under test.

In order to ensure that tool functionality is added in an unobtrusive manner that does not affect regular model operation, has a minimal impact on performance, and provides access to useful functionality, Imperas has created a unique "Binary Interception" concept. This makes use of the Imperas ToolMorphing™ technology to allow tools to be included in the simulation code-morphing operation.

Binary Interception Libraries may be loaded that are then called by the simulator on specified events. These may include:

- Simulation construction and destruction
- Before an instruction is morphed
- When a specific instruction type is executed
- After a certain number of instructions have been executed
- When a specific address or address in a specific range is accessed
- When a specific Programmers View* of a model is executed
- When a command from the library is executed

Once the library is called, it may use an API to query the simulation state in a number of ways, including:

- Examine the processor state including all registers
- Examine the simulation environment
- Replace the simulated behavior of an instruction
- Examine Programmers View* objects
- Examine symbolic and debug information for operating applications
- Use GDB to evaluate expressions within an operating application
- Add or delete callbacks from the simulator to the library

* A Programmers View is a view of a processor or other model created for programmers, which can include specific objects and their states within the model, for example the set of application software accessible registers and/or component internal (hidden) registers.

These calls and operations provide everything needed to support a rich set of custom Verification, Analysis and

Profiling functions. Furthermore, an intercept library may be provided with a specific processor or platform model to enable a range of custom analysis for that model.

Figure 8 shows the simulation execution flow with the binary interception libraries, and how the library contents interact with the model and simulator. This allows the tool code to be morphed with the model and software code during simulation, while still preserving the unobtrusive nature of operation.

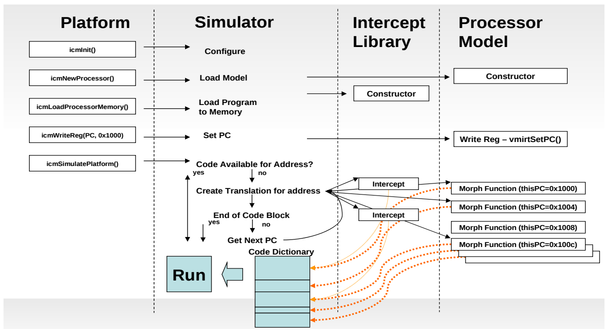


Fig. 8. Simulation Execution Flow With Binary Interception

A broad range of tooling may be created using this method. For software development (unlike hardware), it is very natural for engineers to create powerful capabilities tied to their development programs. With binary interception libraries being able to call functions contained in other libraries, engineers have full access to all the spatial, temporal and abstract commands available and can build functions that leverage output from these facilities and extract exactly what they need. It is this level of functionality that makes this solution so useful for complex embedded software development.

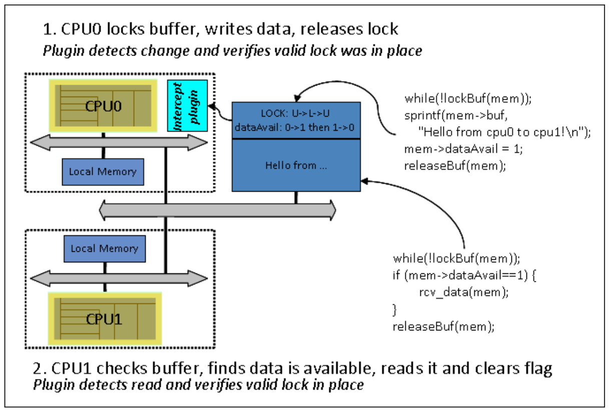
In addition to creating tools, the same solution may be used to interface to other verification systems. For example, the Cadence (formerly Verisity) Specman hardware verification environment was integrated with the OVP simulator to provide a random testbench capability for use with a platform IA model [13]. Integrations have also been performed with hardware debug environments to provide hardware/software co-debug solutions and emulation systems such as Aldec's HES FPGA based emulator [14] all using this method.

VIII. SCEANRIO ONE: NON-INTRUSIVE VERIFICATION FOR SHARED MEMORY MONITORING

A classic source of problems is the use of shared memory in a multicore SMP system, usually the L2 cache, where cache swaps may be scheduled concurrently. Ensuring that the memory is always accessed correctly and that only the permitted memory is accessed in a multicore environment can become extremely complex.

In this specific platform implementation, many threads can access the memory at anytime, so a semaphore solution was used to ensure that memory accesses did not coincide. A protocol was created where the semaphore would be locked during access and a "data available" bit used to indicate data waiting for a read.

To detect misuse of the specified memory region an assertion must be placed in the platform model or the source code, as either could cause operation disruption, but this means changing code of verified models, is always present and is likely to interfere with the normal operation of the system. Instead a memory monitor based on the binary interception technology was used, which on the allocation of the shared buffer placed watch points on the semaphore, see figure 9. Callbacks were then used to monitor the accesses to the shared memory region and create a temporal assertion. Having created this custom, but yet reusable, debug routine the assertion state machine could be augmented with other code to check the access protocol and even some coherency checks.


1. CPU0 locks buffer, writes data, releases lock
Plugin detects change and verifies valid lock was in place

```
while(!lockBuf(mem));
sprintf(mem->buf,
   "Hello from cpu0 to cpu1!\n");
mem->dataAvail = 1;
releaseBuf(mem);
```

```
while(!lockBuf(mem));
if (mem->dataAvail==1) {
      rcv_data(mem);
}
releaseBuf(mem);
```

2. CPU1 checks buffer, finds data is available, reads it and clears flag
Plugin detects read and verifies valid lock in place

Fig. 9. Custom Shared Memory Checker

By using the simulation services, these and other routines could be attached to the model but not interfere with the model code. They could be customized without risking the integrity of the model itself, and perform hardware checks without the need for detailed hardware models. There was no fear of the checks changing the execution of the program, and the impact on performance was minimal. In the real case this solution also lead to the discovery of a bug in production software.

The checks could also be augmented with a range of other functions, for example:

- Print all calls to the buffer lock variable to commence causal analysis
- Using OS Aware tracing, check scheduling sequence and other issues
- Inspect all L1 cache activity to understand if a cache swap problem occurred
- Compare the address space for repeated similar cache activity

All of these functions can be semi-automated and use introspection.

IX. SCENARIO TWO: FAULT INJECTION FOR FAILSAFE OPERATION AND COVERAGE ANALYSIS

Failsafe embedded software is becoming more important for several applications. In the automotive sector, the ISO 26262 standard [16] now specifies rules which safety critical software must meet. For example, software used in an activation chain for the car throttle must fail in such a way that the throttle does not stick. Testing over all possible fault conditions is mandatory.

Changing the software itself to create a test fault does not meet the failsafe test criteria, as the code under test must be the same as the final code base. The fault must be inserted independently of the system. Fault Injection through the simulator solves these problems.

Faults may be injected during operation to create significant issues such as a processor platform locking up, to ensuring more subtle problems do not cause safety hazards. For example, consider an engine management system that uses a lookup table to calibrate throttle pressure based on engine parameters such as temperature. A hardware fault causing the table memory to be addressed incorrectly can be set up by injecting an alternative data fetch instruction just as the throttle pedal is depressed. This test could be repeated for all address bits and the throttle depression analyzed for unsafe behavior. Figure 10 shows an example fault injection sequence output showing the original and replaced instruction and address location.

```
FAULT 0x001002a4(   3004905): 0x0000580d -> 0x00005805(^bit= 3) (        mov) 16 Bits
FAULT 0x00100b8c(  12312824): 0x000059e8 -> 0x000059f8(^bit= 4) (        cmp) 16 Bits
FAULT 0x00100b8c(  17912768): 0x000059f8 -> 0x00005bf8(^bit= 9) (        cmp) 16 Bits
FAULT 0x00100b7e(  19676187): 0x00006f0c -> 0x00016f0c(^bit=16) (       ld.b) 32 Bits
FAULT 0x00100b86(  26529726): 0x00006f4c -> 0x20006f4c(^bit=29) (       st.b) 32 Bits
FAULT 0x00100b7a(  34399330): 0x00006007 -> 0x00006087(^bit= 7) (        mov) 16 Bits
FAULT 0x00100b8e(  38537367): 0x0000f5ea -> 0x0000f5e2(^bit= 3) (          b) 16 Bits
```

Fig.10. Fault Injection Used to Analyze a Hardware Fault Result

In this example, a trap was set on a read to the lookup table area in RAM. The instruction was modified with the same call but with one bit of the address masked out. A coincident checker was set on the throttle driver to ensure that its behavior met the safety criteria laid down in the specification. This test could then be repeated for all address bits. The entire scenario could easily be expanded to allow for many such tests on the throttle behavior.

Fault injection can also be used to establish an effective coverage metric, using mutation analysis. The basic idea of mutation-based coverage [17] is to alter some aspect of the code during runtime, to provoke an error. The error should be picked up during verification if the tests are effective enough to find it. By rerunning the tests with a range of mutations, a coverage metric of the test may be established and test holes identified.

This could be partially accomplished by modifying the source code, recompiling and rerunning the tests. However, this is error prone, time consuming, and functionally restrictive. If a mutation can be introduced separately without a change in the source code, it becomes easier to manage and augment with more powerful functionality.

Returning to our previous example, the throttle control code will have been verified using a standard set of tests. A mutation coverage solution would then rerun these tests, while injecting faults through the design. In this case a systematic replacement of instructions with stuck at 1 or 0 faults would provide an initial set of cover points for further investigation.

X. SUMMARY

The verification requirements for modern embedded software development targeting multicore platforms demand more sophisticated methods. As with hardware design, a simulation-based solution provides an enhanced level of access to both platform model and software detail, while also ensuring execution consistency and minimal performance impact. Simulation is ideal as the core technology for a multi-dimensional framework of verification and debug tools, as well as the capability for custom, platform and OS specific tooling to be developed.

REFERENCES

[1] OMAP Datasheet, Texas Instruments, May 2013 http://www.ti.com/lit/ds/symlink/omap5432.pdf

[2] Symmetric Multiprocessing, Princeton University, 2011 http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Symmetric_multiprocessing.html

[3] High Performance or Cycle Accuracy, SemiWiki, January 2013 https://www.semiwiki.com/forum/content/1979-high-performance-cycle-accuracy-you-can-have-both.html

[4] Accellera, SystemC Standard http://www.accellera.org/community/systemc

[5] QEMU, Standard Main Page http://wiki.qemu.org/Main_Page

[6] Open Virtual Platforms http://www.ovpworld.org

[7] OVP API Description http://www.ovpworld.org/technology_apis

[8] Heisenbug Definition http://catb.org/jargon/html/H/heisenbug.html

[9] GNU GDB Reference Manual http://sourceware.org/gdb/current/onlinedocs/gdb/

[10] O'Reilly, 2003, Introspection Definition http://docstore.mik.ua/orelly/webprog/php/ch06_05.htm

[11] Accellera UVM Standard Description, 2010 http://www.accellera.org/activities/committees/uvm

[12] OVP MIPS Malta Platform Description 2010 http://www.ovpworld.org/library/wikka.php?wakka=Mips32MaltaLinux

[13] Synopsys Verdi Hardware Debug Datasheet https://www.synopsys.com/Tools/Verification/debug/Pages/Verdi-ds.aspx

[14] Posedge Integration of Specman with OVPSim, July 2009 http://www.ovpworld.org/resources/VP09-content/posedge_von_bank_vp09.pdf

[15] Proximus Aldec Emulation Integration, Design Automation Conference, June 2011 http://www.ovpworld.org/aldec-cadence-proximus-utilize-ovp-fast-processor-models-in-system-design-solutions

[16] ISO 26262 Road Safety Standard, 2012 http://www.iso.org/iso/search.htm?qt=26262&sort_by=rel&type=simple&published=on&active_tab=standards

[17] An Analysis and Survey of the Development of Mutation Testing, IEEE Transactions on Software Engineering, vol. 37 no. 5, pp. 649 – 678, September 2011