Customized, Intelligent Memory Access Monitoring for Reliable Asymmetric MultiProcessor System Development

Simon Davidmann and Larry Lapides
Imperas Software Ltd.
Oxford, United Kingdom
larryl@imperas.com

Stefano Zammattio
Altera Corporation
High Wycombe, United Kingdom

***Abstract* – The use of Asymmetric MultiProcessor (AMP) architectures is now widespread. Two common implementations are Linux running on one core of a dual-core ARM Cortex-A9, with an RTOS running on the other, and SMP Linux running on the dual-core ARM Cortex-A9 and an RTOS or bare metal application running on another processor core, such as an Altera NIOS II. The reliability of such a system is highly dependent on the correct functioning of inter-core interaction with shared resources, which is often hard to verify. Similar issues with reliability and verification are encountered when extending such a system to include security, such as utilizing the ARM TrustZone instructions.**

Many embedded software verification issues limitations are related to the lack of visibility and controllability inherent in the use of hardware platforms for software testing. Moreover, without this visibility and controllability the hardware-based analytical tools require the instrumentation of the source code or the use of a special "debug" compiled version of the OS, which by their very nature perturb the system and reduce the validity of the analysis results. Virtual platforms (software simulation) provide a test vehicle which, with the right methodology and tools, can provide the visibility and controllability needed for comprehensive testing of embedded software on AMP systems.

This paper will detail the methodology used to bring up such an AMP system on a virtual platform. The first step is the construction of the instruction accurate virtual platform for the two AMP systems. The Open Virtual Platforms APIs for model and platform development are used in building the virtual platforms used in this paper. These virtual platforms are variants of the Altera SoC FPGA Cyclone V product. The second step in the process involves the use of CPU- and OS-aware analysis tools to help with initial system bring up. Rather than providing only instruction trace data, these tools enable the analysis of the system at the appropriate level of abstraction for the software engineer: C source code for firmware and drivers and the OS task/event level for operating systems. In addition, the tools are non-intrusive, requiring no instrumentation or modification of the application or OS, thus validating the results of the analysis.

Finally, the third step is the development of a robust test environment, including the use of non-intrusive, intelligent memory access monitors, built upon the CPU- and OS- aware simulation environment to ensure that different OS operations do not access forbidden memory segments.

A detailed case study illustrating how complex faults have been found in an Altera Cyclone V AMP system, by using this methodology, will be shown.

Keywords – Virtual platform, AMP, memory monitor, OS-aware, software tools

I. INTRODUCTION

As embedded systems have become more complex, different operational architectures have evolved, having to do with the multiple processors available in a single silicon device and systems of multiple silicon devices. These silicon devices, because of the increasing complexity and multiple processors, have become known as Systems on Chip (SoCs). The multiple processors on a given SoC can be fully homogenous, or heterogeneous. Combined with the different processors are different operating systems, either Symmetric MultiProcessor (SMP) or Asymmetric MultiProcessor (AMP) [1]. The combinations of processor and operating system are shown in Table 1.

Table 1. Different multiprocessor operating architectures for embedded systems.

	Homogeneous Processors	**Heterogeneous Processors**
Symmetric MultiProcessor	Homogeneous SMP	Not done in practice
Asymmetric MultiProcessor	Homogeneous AMP	Heterogeneous AMP

In SMP systems, a single operating systems (OS) runs on all the cores. This OS manages all the resources, making it appear to the user that there is just one core. Most of the complexity is hidden from the user. This ease of use is nice, but the performance advantages of having multiple cores are more difficult to achieve.

In AMP systems, different OSs run on different cores. For example, Linux could run on one core, and a Real Time Operating System (RTOS) on another core, enabling the user to

employ the optimal OS for different tasks. Parallelization of applications is easier, resulting in better performance gains. However, the user needs to manage the resources, so this is not as easy to implement in practice. AMP systems also are used with multiple, heterogeneous cores so that not only is the OS optimized, but the OS-processor core combination is optimized for different tasks.

For a SMP system, the user just needs to port and bring up the SMP OS on the SoC. This is not a trivial task, however, with the Linux community there is a lot of help available. For an AMP system, multiple OSs need to be ported and brought up on the SoC, at least one of which will be a RTOS. Also, since the user has to manage the resources and communication, there is more opportunity for problems to arise. A well thought out methodology is necessary to ensure that the AMP system will work as designed.

In this paper the process for bring up of an AMP system is described. The methodology is based on the use of virtual platform technology, starting with the initial development of the virtual platform and bring up of Linux on a single core and finishing with bring up of the full AMP system. The steps are described, showing how operating system (OS) aware tools and customized memory access monitors can play a significant role in easing the bring up process, reducing schedule and achieving higher quality software. The current hardware-based software development flow is first discussed, followed by a general discussion of virtual platform technology, before describing the AMP bring up methodology.

II. Limitations of Hardware-Based Software Development, Debug and Test

The standard methodology for embedded software development is to use some type of hardware as the development platform. This could be a previous generation of the SoC, a hardware emulator, a FPGA prototype, or some other type of development board. These platforms have the benefit of cycle accurate execution of the software, which is needed for some software development. However, the requirement for cycle accurate development platforms is overstated by users.

While there are advantages to using a hardware-based development methodology, there are also disadvantages. These disadvantages include

- Limited physical system availability
- Limited external test access (controllability)
- Limited internal visibility
- Typically 6 months or more from project start until hardware platform is available to software engineering team

To get around these limitations, the software is typically modified for debug purposes. This takes the form, for example, of adding printf and printk commands, using debug versions of the OS kernels, and adding instrumentation to the source code in order to use analytical tools. In all these cases the software is modified, and the perturbed software will almost certainly not have the same behavior as the clean source code. This results in additional difficulties in debug, and puts the validity of analytical results in doubt.

III. Software Simulation (Virtual Platform) Advantages

Instruction accurate virtual platforms, which are just software simulation environments, are not cycle or timing accurate. However, these virtual platforms do have significant advantages:

- Earlier system availability
- Full controllability of platform both from external ports and internal nodes
- Full visibility into platform
- Performance can be faster than real time
- Easy to replicate platform and test environment to support regression testing on compute farms

Looking at the complete software development methodology, virtual platforms should be used exclusively early in the development process. However, the virtual platforms, due to the visibility and controllability and the software development tools available, can continue to be used and add value throughout the duration of the software project. The hardware platforms, including the final hardware, can be used when they become available, with hardware based testing and virtual platform based testing providing complementary benefits to software engineers.

Before going further, some additional understanding of the details of a virtual platform is needed.

Begin with an example of the software architecture to run on a SoC, in this case for picture in picture video (Figure 1). This architecture is then translated via manual engineering effort into the hardware implementation. Conceptually, the hardware implementation can then be translated into high level models of the hardware components, including the various processors (Figure 2). In practice, once the high level hardware implementation is architected, the virtual platform models are usually developed well ahead of the detailed hardware implementation. As with Intellectual Property (IP) models for hardware design, high level models of standard components should be readily available from existing libraries.

For instruction accurate virtual platforms, the models should have only as much information in the models as the software developers need. There is no timing information needed in these models, and any unnecessary information will slow the virtual platform performance. The processor models can be thought of as similar to instruction set simulators. However, for virtual platforms, the ideal is to have a single simulator for both the

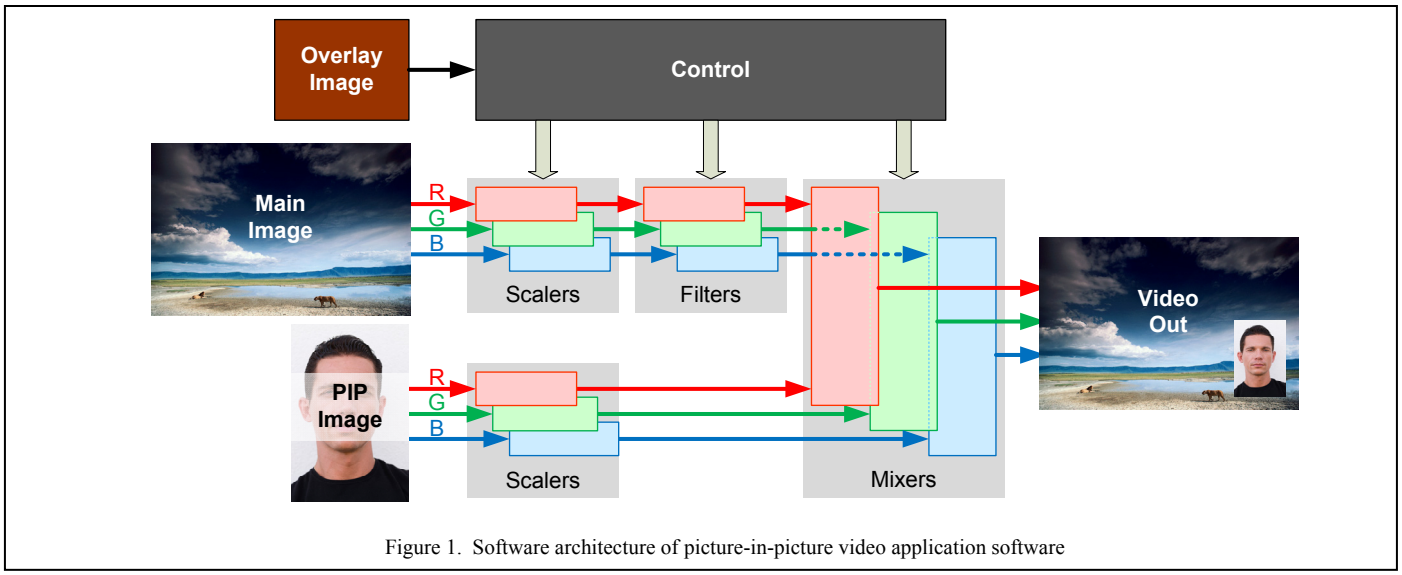

Figure 1. Software architecture of picture-in-picture video application software

processors and the peripheral and behavioral components, and not a difficult-to-use and low performing co-simulation of processors and other components. Architecturally, this means that the models are separate from the simulator engine. While this is advantageous from a performance and ease-of-use perspective, it also enables software tools to be easily and efficiently added into the simulation environment.

One last key point about virtual platforms is that the combination of virtual platform models plus simulator executes exactly the same binary software stack as will eventually run on the hardware. No compiling for the host x86 workstation; if the system uses an ARM processor, the same cross compilation tool chain and flow are used to create the ARM binary executables to run on the virtual platform or the hardware.

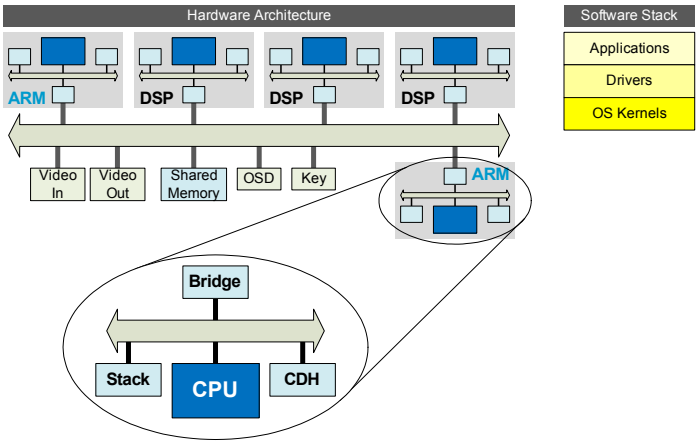

Figure 2. Hardware implementation of picture in picture video system, also showing the software stack that would run on the SoC. The virtual platform implementation of this system looks exactly the same conceptually, but is composed of high level, instruction accurate models of the hardware.

IV. Case Study: OS Porting, Bring Up and Verification on Altera Cyclone V SoC FPGA

A. *Altera Cyclone V Device*

The Altera Cyclone V SoC FPGA [2] was used because of its power and versatility. This device combines the power of an ARM-based SoC with the versatility of a FPGA. Key features of the Cyclone V include a dual core ARM Cortex-A9 processor (Cortex-A9MPx2), a rich set of peripheral components in the Hard Processor System (HPS) including ethernet, USB, I2C, and more, and a large amount of FPGA fabric that the user can utilize for custom functionality. Altera offers a number of components as IP for the fabric, including the Nios II processor core. A block diagram of the Cyclone V HPS is shown in Figure 3.

One of the advantages of the Cortex-A9MPx2 is that it can be configured to run in either SMP or AMP modes. Also, for AMP operation, the Cortex-A9MPx2 can be running in SMP mode, and one or more Nios II processor cores can be added to the fabric for heterogeneous AMP operation. Both scenarios are discussed below.

B. *Open Virtual Platforms Models and Modeling APIs*

Open Virtual Platforms (OVP) [3] models and modeling APIs were used to build the virtual platform of the Cyclone V because of the availability of the processor core models in the OVP library, the performance of the processor core models, the ease of use of model development with the APIs and the tool power that is enabled when running the platform with the Imperas simulators.

OVP includes a library of models, APIs for developing models and platforms and a reference simulator (OVPsim) for executing those virtual platforms. The model library includes over 125 different processor core models, plus over 100

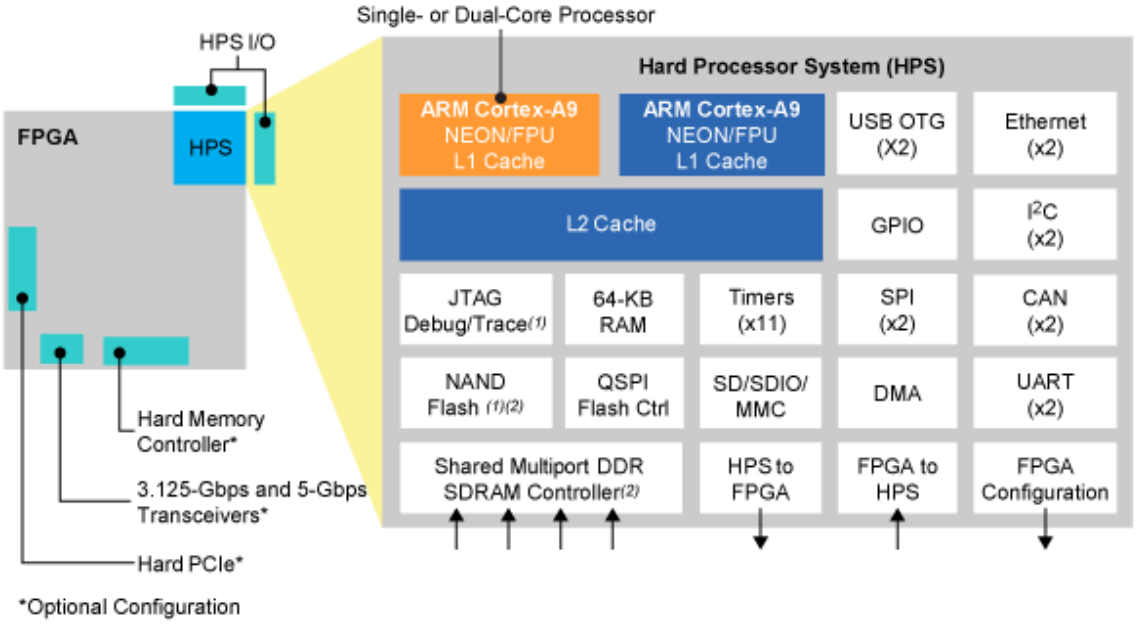

Figure 3. Cyclone V SoC FPGA Hard Processor System

different peripheral models, and a variety of example virtual platforms. While most of the models available from the OVP website have been developed by Imperas, other users have contributed models to the library. Most models are available as both binary and source, with distribution governed by a modified Apache 2.0 open source license.

C. Linux Boot on Single Core ARM Cortex-A9

The first step OS porting and bring up was to boot Linux on a virtual platform containing only a single core ARM Cortex-A9 model. One of the advantages of using a virtual platform for software development is that the virtual platform in total, as well as the individual models, do not need to be complete. By this it is meant that the virtual platform does not need to include models of all the components on the hardware device, and some of the component models may only include a portion of the functionality or may only include the top level registers.

This methodology was used to create the initial Cyclone V virtual platform, shown in Figure 4. Table 2 lists the components in the virtual platform, and the degree of accuracy or completeness of the model.

Table 2. Virtual platform component models.

Model	Model Development	Completeness
ARM Cortex-A9	OVP Library item	Complete functional model
DMA	OVP APIs	Registers only
Ethernet	OVP APIs	Registers only
Imperas SmartLoader	OVP APIs	Complete functional model
SRAM	OVP APIs	Complete functional model
System Manager	OVP APIs	Registers only
Timers	OVP APIs	Complete functional model
UART	OVP APIs	Complete functional model

Development of the individual models and construction of the virtual platform from the newly developed models and from models from the OVP library took about 2 weeks of effort to achieve a Linux prompt. Linux version 3.4 from Altera was used for single core Linux. Default configurations were used. The device tree was modified to comment out peripherals that were not included in the virtual platform, as those peripherals were not critical to the booting and testing of Linux. The time to Linux prompt, the "virtual platform boot time", is under 5 seconds for this virtual platform running on a standard x86 Windows or Linux host machine.

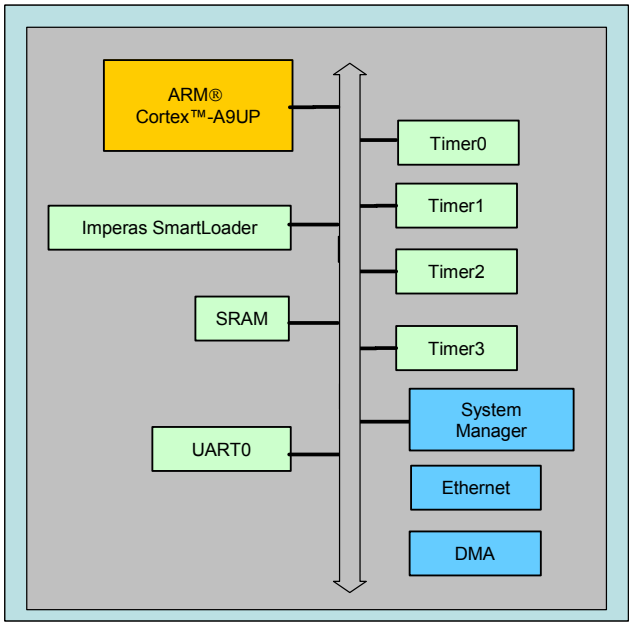

Figure 4. Virtual platform of the Altera Cyclone V used to boot Linux

Once Linux was booting on the virtual platform, a suite of tests was run to validate Linux behavior on the platform. In the course of running this suite of tests, a bug was found in the Linux kernel preemptive scheduling: Linux did not switch correctly between multiple tasks, corresponding to multiple applications, running in the Linux OS. OS-aware tracing tools were used to find the bug, which involved reading from the incorrect register in one of the timers. Once the bug was observed, finding and fixing the bug took only hours.

D. Software Development, Debug and Test Tools

Virtual platforms offer the advantages of controllability and visibility, but what tools are available to enable the realization of these advantages? Using the Imperas virtual platform environment, a suite of tools is available for use, plus users can define custom tools. These tools use the binary interception capability built into the Just In Time (JIT) code morphing [4] simulation engine. Binary interception works by intercepting the processor instructions before execution by the JIT engine. Additional code may be added to the instruction for analytical purposes, and then the combined instruction is processed by the JIT engine. The original processor instruction is still executed, unchanged, but now the additional analysis instruction is also executed. One advantage over other virtual platform based analysis techniques is that by using the JIT engine, the performance overhead for any analysis tools is minimized, typically achieving less than 10% [5].

Using binary interception for building software analysis tools results in enabling analysis without modifying the software – OS, firmware, driver, application – source code, a significant advantage over hardware based software development environments. In these environments, source code must be instrumented and recompiled, such as for gcov and gprof, or a debug version of the kernel must be used, or printf or printk instructions are liberally sprinkled through the source code to enable debug. These actions change the behavior of the software, and as a result the bugs found and fixed in this scenario are not necessarily what the user was trying to find.

A C language API [6] has been developed which enables tool builders and end users to take advantage of the binary interception capability. "Intercept libraries" are compiled for the host machine, and then added in to the simulation environment. CPU- and OS-aware tools have been built which raise the abstraction level for analysis. As a simple illustration of the value of these tools, debugging Linux bring up with instruction tracing would require sifting through about 700,000,000 instructions before the boot prompt was observed. However, OS task tracing reduces this to the approximately 700 tasks that are executed as Linux boots. This makes it much easier to narrow down the cause of any problem. Figure 5 shows a screen shot of OS-aware tracing of task, execve and scheduler as Linux boots.

E. SMP Linux Boot on Dual Core ARM Cortex-A9

Once Linux was booting on the single core virtual platform, the single core ARM processor model was replaced by the model of the dual core Cortex-A9. Another UART was added to the virtual platform for the second core, and an L2 Cache Controller model with only the necessary functionality was also added. Then SMP Linux was executed on the virtual platform. Linux version 3.6 from Altera was used. There were no problems bringing up SMP Linux on the virtual platform.

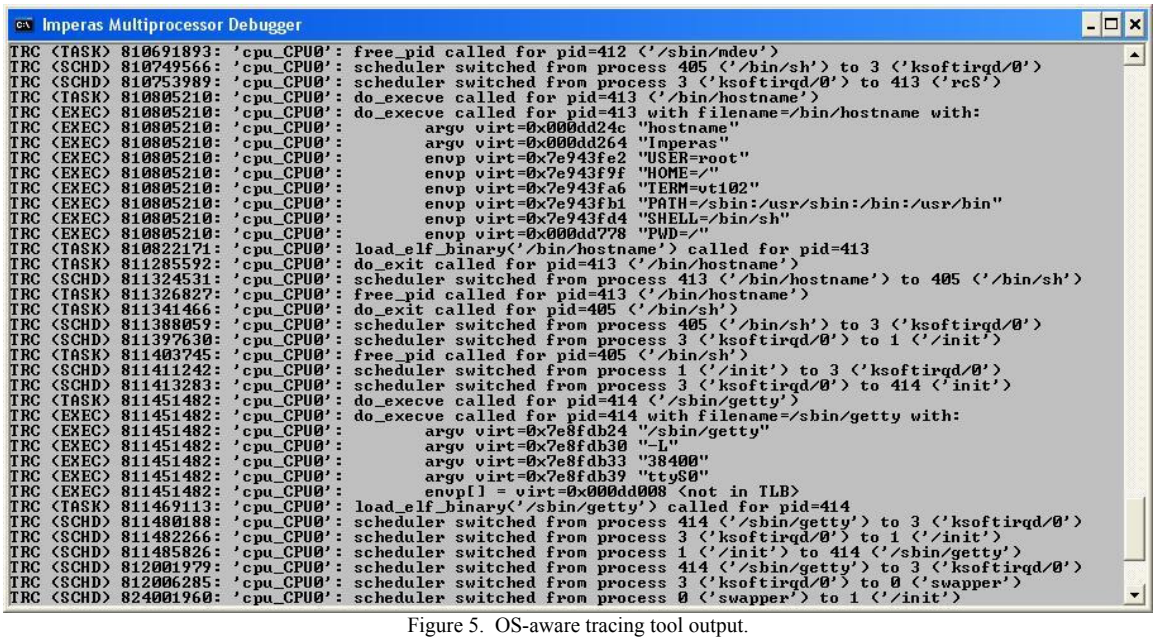

Figure 5. OS-aware tracing tool output.

F. RTOS Boot on Single Core ARM Cortex-A9

The third step in the process was to bring up the RTOS on a single core ARM Cortex-A9 platform. This just reused the initial platform used in the first step for single core Linux bring up. The RTOS used was the Altera release of the Micrium µC/OS-II [7] RTOS.

When running example applications under the RTOS, incorrect behavior was observed. Again using the OS-aware tools, this time tuned for µC/OS-II, bugs were discovered in the global interrupt controller (GIC) register accesses. These were easy to find using the tools; in contrast the bugs had not been observed when bringing up the RTOS on hardware. The easy-to-fix bugs were

- Accessing ICDICER 1 to 8 when only 0 to 7 exist
- Accessing ICDIPTR 08 to 63 when only 00 to 55 exist

G. AMP Boot on Dual Core ARM Cortex-A9

The fourth step was to bring up an AMP system involving just the dual core ARM Cortex-A9. In this case, Linux would run on one core, and µC/OS-II on the other core. This is not too different from each operating system running on a single core. However, the operating characteristics are more complicated, including the requirement to ensure that each operating system does not access memory that is dedicated to the other operating system.

For this step, the virtual platform needed to be modified to add the Altera Reset Manager component. Only the minimal functionality needed to handle the reset prior to each OS booting was required for the model. All other components remained the same. Also, the Linux and µC/OS-II OS-aware tools were used in the simulation environment. The additional tool added to the simulation environment at this step was a custom memory access monitor.

This custom memory access monitor used the binary interception API to define which memory regions could be accesses by which of the operating systems. This took the form of a table written in C:

```
//
// Define watch areas for memory and peripherals defined in the platform
//
memWatchT amcWatch[] = {
//  name              watchLow    watchHigh allowedCPUs
    { "Linux memory",   0,          0x2fffffff,   LINUX_CPU   },
    { "uCOS memory",  0x30000000, 0x31ffffff,   UCOSII_CPU  },
    { "gmac0",        0xff700000,  0xff700fff,  LINUX_CPU   },
    { "emac0_dma",    0xff701000,  0xff701fff,  LINUX_CPU   },
    { "gmac1",        0xff702000,  0xff702fff,  LINUX_CPU   },
    { "emac1_dma",    0xff703000,  0xff703fff,  LINUX_CPU   },
    { "uart0",        0xffc02000,  0xffc02fff,  LINUX_CPU   },
    { "uart1",        0xffc03000,  0xffc03fff,  UCOSII_CPU  },
    { "CLKMGR",       0xffd04000,  0xffd04fff,  LINUX_CPU   },
    { "RSTMGR",       0xffd05000,  0xffd05fff,  LINUX_CPU   },
    { "SYSMGR",       0xffd08000,  0xffd08fff,  LINUX_CPU   },
    { "GIC",          0xfffec000,  0xfffedfff,  LINUX_CPU   },
    { "L2",           0xfffef000,  0xfffeffff,  LINUX_CPU   },
    { 0 } /* Marks end of list */
};
```

In the "Allowed CPU" column, "Linux CPU" and "UCOSII CPU" were used instead of CPU0 and CPU1 for clarity.

When the virtual platform was run, there were two results. First, a bug was found in the Linux accesses of the GIC registers. This bug had been previously found in a hardware bring up of the AMP system. Working on hardware, it took about 2 weeks to find and fix the bug. In contrast, using the virtual platform with the OS-aware tools took only 2 days to find and fix the bug.

The second result was from the memory access monitor, in the form of warnings about unallowed memory accesses. A partial list of those warnings is

```
Warning (AMPCHK_MWV) cpu_CPU0: AMP write access violation in uart1 area. PA: 0xffc03008 VA: 0xffc03008
Warning (AMPCHK_MWV) cpu_CPU0: AMP write access violation in uart1 area. PA: 0xffc0300c VA: 0xffc0300c
Warning (AMPCHK_MWV) cpu_CPU0: AMP write access violation in uart1 area. PA: 0xffc03010 VA: 0xffc03010
Warning (AMPCHK_MRV) cpu_CPU1: AMP read access violation in Linux memory area. PA: 0x00000020 VA: 0x00000020
```

Note that both physical and virtual memory address are given as part of the warning message. Using these messages, issues with the low level boot code were quickly fixed, resulting in safe operation of the AMP system.

H. Linux Boot on Single Core Nios II

For this step, a simple virtual platform with the model of the Altera Nios II [8] processor was used (Figure 6). The same OS-aware tools that were used earlier for other Linux bring up steps on ARM were used again for the Nios II Linux bring up, as these tools are not specific to the processor core. Using this

process enabled quick bring up of Linux on the Nios II virtual platform.

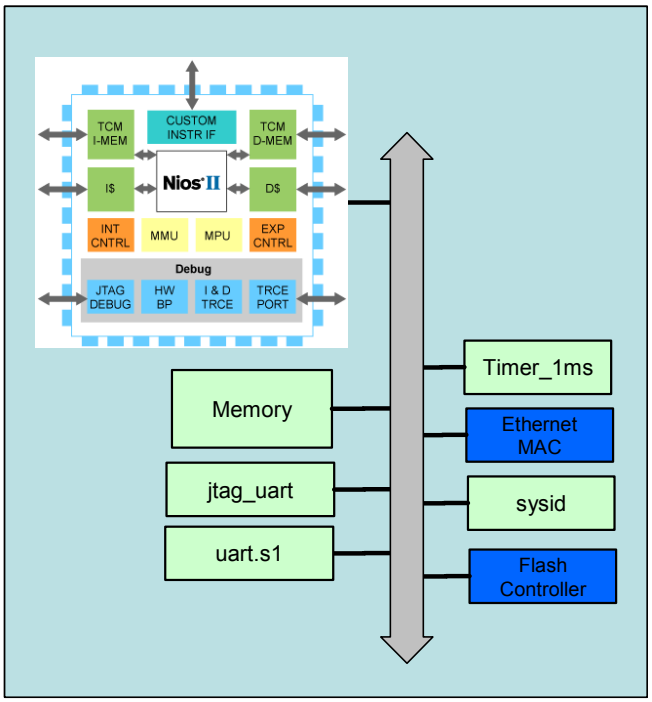

Figure 6. Virtual platform used for booting Linux on Nios II

I. Full AMP System: SMP Linux Boot on Dual Core ARM Cortex-A9 Plus Linux Boot on Nios II

The final step was combining the SMP Linux ARM Cortex-A9MPx2 virtual platform with the Nios II Linux virtual platform, and running these together. Due to the work performed in the previous steps, there were no issues in this final step.

V. SUMMARY

As embedded systems get more complex, software testing becomes more critical. Software testing on hardware systems has limitations in the controllability and visibility that can be detrimental to comprehensive testing of the software. Instruction accurate software simulation – virtual platforms – complement hardware based environments because of having the required controllability and visibility. Virtual platform based OS-aware and custom memory access monitoring tools can provide both higher quality and reduced schedules for OS porting and bring up. Results were shown for a virtual platform of a heterogeneous ARM-Nios AMP system on Altera Cyclone V SoC FPGA device.

REFERENCES

[1] "Symmetric Multiprocessing Vs. Asymmetric Multiprocessing", Toby Foster, Electronic Design Magazine, 13 December 2007 (http://electronicdesign.com/digital-ics/symmetric-multiprocessing-vs-asymmetric-processing).

[2] "Cyclone V Device Overview", Altera Corporation, Version 2013.12.26.

[3] Open Virtual Platforms API documentation and user libraries are available at www.OVPworld.org

[4] "Just In Time Compilation", Wikipedia article, http://en.wikipedia.org/wiki/Just-in-time_compilation.

[5] "Embedded Software Dynamic Analysis: A New Life for the Virtual Platform", Victoria Mitchell, presented at the North American SystemC User Group 19 (NASCUG 19) meeting, 3 June 2012.

[6] "Imperas Binary Interception Technology User Guide", Imperas Software Limited, 2013.

[7] µC/OS-II RTOS Overview, http://micrium.com/rtos/ucosii/overview/, Micrium, Inc. 2014.

[8] "Nios II Processor: The World's Most Versatile Embedded Processor", Altera Corporation, http://www.altera.com/devices/processor/nios2/ni2-index.html, 2014.