Instruction-Set-Simulator-less Virtual Prototype Framework for Embedded Software Development

by

Nick Ni

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright by Nick Ni 2011
Instruction-Set-Simulator-less Virtual Prototype Framework for Embedded Software Development

Nick Ni
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2011

ii. Abstract

With continuous advancement in silicon technology and high feature demands on consumer electronics, the complexity of embedded software has led the software development effort to dominate System-On-Chip (SoC) design. Virtual Prototype (VP) addresses the problem by enabling early software development before hardware arrival. However, VP still poses challenges: 1) Instruction Set Simulator (ISS) degrades simulation time, 2) Development is restricted to embedded processor specific tools and 3) Applications and drivers are dependent on system software completion. In this work, we propose an abstraction framework which: 1) Removes ISS from VP, achieving native host software execution time, 2) Activates rich suites of desktop development tools in host by compiling embedded software in host binary and 3) Allows system software independent application and driver development. With this framework, we successfully demonstrated up to 2000% speed-up in VP run-time over conventional VP and improved software development productivity significantly.
Table of Contents

Table of Contents .................................................................................................................. iii
List of Figures .......................................................................................................................... vi
List of Tables ............................................................................................................................ ix
Chapter 1 Introduction ........................................................................................................... 1
  1 Introduction .......................................................................................................................... 1
    1.1 Motivation ...................................................................................................................... 1
    1.2 Contributions and Benefits .............................................................................................. 7
    1.3 Thesis Overview ............................................................................................................. 7
Chapter 2 Background ............................................................................................................. 9
  2 Background .......................................................................................................................... 9
    2.1 SoC design flow .............................................................................................................. 9
    2.2 Virtual Prototype ......................................................................................................... 12
    2.3 QEMU .......................................................................................................................... 16
      2.3.1 Dynamic Translation Based ISS ............................................................................. 18
      2.3.2 Peripheral Model / Host Peripheral Access ......................................................... 19
    2.4 Previous Work .............................................................................................................. 21
    2.5 Summary ....................................................................................................................... 23
Chapter 3 ISS-less Virtual Prototype Framework ..................................................................... 24
  3 ISS-less Virtual Prototype Framework ............................................................................... 24
    3.1 Objectives ...................................................................................................................... 24
    3.2 Key Ideas ....................................................................................................................... 24
    3.3 Framework Architecture .............................................................................................. 27
      3.3.1 I/O Memory API .................................................................................................... 28
      3.3.2 Interrupt API ......................................................................................................... 31
3.3.3 Arch .................................................................................................................. 33

3.4 Network Software Stack Example with RTOS ....................................................... 34

3.4.1 Conventional VP flow .......................................................................................... 35

3.4.2 I/O Memory API Porting ...................................................................................... 37

3.4.3 Interrupt API Porting ........................................................................................... 38

3.4.4 Arch Porting ......................................................................................................... 39

3.4.5 Compiling Contiki ............................................................................................... 41

3.4.6 Modifications to QEMU ...................................................................................... 41

3.4.7 Contiki execution flow on our framework .......................................................... 43

3.5 Summary .................................................................................................................. 44

Chapter 4 System Software Independent Application and Driver Development ........ 45

4 System Software Independent Application and Driver Development ................... 45

4.1 Objectives ............................................................................................................... 45

4.2 Application Process ............................................................................................... 48

4.2.1 POSIX compliant standard library ...................................................................... 49

4.2.2 Threading Library ............................................................................................... 50

4.2.3 Virtual File System ............................................................................................. 52

4.2.4 System Call API .................................................................................................. 53

4.2.5 Arch ..................................................................................................................... 54

4.2.6 User Signals ........................................................................................................ 55

4.3 Emulated Kernel Process ...................................................................................... 56

4.3.1 System Call Poller ............................................................................................... 57

4.3.2 Driver API ............................................................................................................ 59

4.3.3 TCP/IP Stack ....................................................................................................... 60

4.4 Example with H.264 Video Decoding using FFMpeg ........................................... 60

4.4.1 Building Application Process ............................................................................ 61
List of Figures

Figure 1: SoC design cost breakdown for different technology nodes ........................................... 2

Figure 2: An example virtual prototype for a wireless phone ....................................................... 4

Figure 3: Software execution on ISS (left) vs native execution on host (right) .............................. 5

Figure 4: Tools availability for ARM software and desktop (x86) software ................................. 5

Figure 5: Traditional SoC design flow vs Virtual Prototype design flow ..................................... 10

Figure 6: A Virtual Prototype example ....................................................................................... 14

Figure 7: ARM SoC platform built using QEMU ........................................................................ 17

Figure 8: ISS implementations: Interpretation vs Dynamic Translation ..................................... 19

Figure 9: Peripheral Model Access and host peripheral interaction in QEMU .............................. 20

Figure 10: Instruction set independence in embedded software .................................................. 25

Figure 11: Memory-map dependency in embedded software ...................................................... 25

Figure 12: ARM interrupt vector table ........................................................................................ 26

Figure 13: ISS-less Virtual Prototype Framework Architecture Diagram ..................................... 28

Figure 14: I/O Memory API ........................................................................................................ 29

Figure 15: Memory-mapped I/O abstraction illustration .............................................................. 30

Figure 16: Conventional VP interrupt delivery mechanism .......................................................... 31

Figure 17: Interrupt delivery mechanism in ISS-less VP Framework .......................................... 32

Figure 18: Interrupt API .............................................................................................................. 33

Figure 19: SoC platform built in QEMU for demonstration .......................................................... 35
Figure 20: Life time of a packet on Contiki with QEMU .......................................................... 37

Figure 21: Example of I/O Memory API porting in a real-life driver. Left: original driver and
Right: ported to I/O Memory API .......................................................................................... 38

Figure 22: Example of Interrupt API porting in a real-life embedded software stack. Left:
original global ISR. Right: ported to interrupt API .................................................................. 39

Figure 23: Modifications to QEMU. Top: Original QEMU, Bottom: QEMU with ISS-less VP
framework .................................................................................................................................. 42

Figure 24: Life time of network packet with ISS-less VP Framework................................. 44

Figure 25: Usage Model I: Separate processes for applications and drivers ....................... 47

Figure 26: Usage Model II: Single process for applications and drivers.............................. 47

Figure 27: Application Process ............................................................................................... 49

Figure 28: Original newlib (left) vs modified arch/platform independent newlib (right) ....... 50

Figure 29: Threading Library interaction with neighboring layers ........................................ 51

Figure 30: Virtual File System Implementation on Host FS (left) and Network FS using 9P
(right) ........................................................................................................................................ 52

Figure 31: System Call API message with write() system call ............................................. 53

Figure 32: User Signal mechanism in between Application and Emulated Kernel Processes. .... 56

Figure 33: Emulated Kernel Process ........................................................................................ 57

Figure 34: Example of how “gettimeofday” system call is served in Emulated Kernel Process . 58

Figure 35: Driver API .............................................................................................................. 59

Figure 36: Example H.264 video decoding application ........................................................... 61
Figure 37: Life time of an encoded video frame in Network FS until it is decoded and displayed on LCD peripheral model in ISS-less VP ................................................................. 64

Figure 38: Two reference SoC platforms with a: ARM and b. Microblaze ISS vs c. ISS-less VP ........................................................................................................................................ 66

Figure 39: Combined software stack with System Call API directly calling System Call Poller functions ........................................................................................................................................ 67

Figure 40: Target-arm/Target-mb configuration ......................................................................................................................... 69

Figure 41: uninitialized C struct example with sigaction system call .............................................. 88
List of Tables

Table 1: Properties of different simulation methodologies ................................................. 12
Table 2: Example of mapping SoC peripheral models with host peripherals in QEMU .......... 17
Table 3: Summary of related work on speeding up virtual prototype ................................. 23
Table 4: Architecture and Platform dependent components in embedded software ............. 27
Table 5: Non-I/O architecture dependent components in Software Stack ............................. 34
Table 6: Arch porting for Contiki ......................................................................................... 40
Table 7: Arch directory contents ......................................................................................... 54
Table 8: Major system call dependencies ........................................................................... 58
Table 9: Host Specification ................................................................................................. 67
Table 10: Different configurations of software stack ............................................................ 68
Table 11: Benchmark description and statistics .................................................................... 71
Table 12: GZIP performance .............................................................................................. 74
Table 13: Frame Buffer Write Performance ......................................................................... 77
Table 14: File Access Performance ..................................................................................... 80
Table 15: Real-life application performance ....................................................................... 83
Table 16: SPEC2000 bench mark performance ................................................................... 85
Table 17: Tasks for implementing the Arch library for ARM and Microblaze ...................... 86
Chapter 1
Introduction

1 Introduction

1.1 Motivation

Not only has continuous advancement in fabrication technologies pushed the complexity of System-On-Chip (SoC) to historical high with billions of transistors on a single chip [2], but also success of feature-rich consumer electronics such as iPhone and Blackberry has triggered ever increasing demand on software features and performance on SoC. Delivered SoC today is not designed solely to serve any single or particular set of applications, but rather to leverage system software / hardware combinations to adapt to continuously evolving user needs. This trend is pushing system and SoC vendors to deliver complete software stacks along with their system software such as firmware, drivers and Operating System (OS). As a result, the costs of software development (even with software re-use) have surpassed the costs associated with hardware and have become the largest and fastest growing portion of total SoC design as shown in Figure 1 [30].
Figure 1: SoC design cost breakdown for different technology nodes

Industries have long identified that traditional hardware-centric SoC design approach where hardware was first designed and given to software team for start of development, no longer works due to increasing engineering time in software. It soon became clear that a hardware/software co-development framework is a key to success of a project as it allows software team to begin prototyping and development before the hardware board has come back from fabrication facilities. Virtual Prototype (VP) is one of the commonly used frameworks in industry for this purpose. It is a set of fully functional software models of SoC hardware such as processors, peripherals and bus. Processor model is often implemented using Instruction Set Simulator (ISS) which provides binary compatibility with embedded processor (called Target) instruction set. ISS converts target instruction set to instruction set of the desktop PC running the simulation (called Host) to allow execution of un-modified embedded software. Bus and
peripheral are typically modeled using a high-level language such as ANSI C or SystemC [6] with focus on pin-accurate software visible interfaces such as register, bus protocol and peripheral functionality. VP typically runs on an off-the-shelf PC host and it is trivial to provide a copy of complete SoC platform to every member of SoC project team. Using VP as their hardware platform, software engineers are able to start their work while hardware engineers are designing Intellectual Properties (IP) and integrating with SoC. Figure 2 shows a typical wireless phone SoC design implemented in VP. It contains ARM ISS and DSP ISS which run production embedded software stacks. Bus communications and peripheral devices are implemented using behavioral models with accurate interfaces to both ISS to provide software illusion of existence of real bus and peripherals. The details of the bus and peripheral models range from slow cycle-accurate model to fast untimed functional model with trade-off of architectural details. For majority of software development, because timing is not a concern, only software visible register interfaces, bus protocols and peripheral functionalities need to be modeled. Due to its highly abstracted architectural modeling, VP is able to run the complete embedded software stack and boot OS in tens of seconds, making it suitable for software development. VP can also support real-time software which requires certain timing behavior such as interrupt timing, by adding timing accuracy to the models. However, this tends to slow-down the VP execution time significantly.
While VP offers significant advantages to software team, it also poses several challenges.

1. **ISS slows down execution time.** Different implementations of ISS such as interpretation and dynamic translation have very contrasting performance. Interpretation-based ISS can result in magnitude of three slow-downs compared to native software execution [29]. Dynamic translation drastically improves the ISS performance up to 50% of native software execution often with sacrifice of cache and pipeline simulation [41]. However, because both implementations require an extra step to convert target instruction set to host instruction set, ISS tends to be slower than native software execution on host (Figure 3).
2. *Development environment is limited* to target processor specific tools. Because the software under development is cross-compiled for target processor, only tools built for the target architecture can be used. The availability of such tools is much more limited than wide range of tools available for desktop environments (Figure 4) such as memory analysis tools (e.g. Valgrind [14]) and formal verification tools (e.g. Perfect Developer [15]). Porting tools such as GNU compiler and debugger for the target also incurs non-trivial engineering time and tends to be very error-prone until time proves its tool maturity.

![Software execution on ISS (left) vs native execution on host (right)](image)

Figure 3: Software execution on ISS (left) vs native execution on host (right)

![Tools availability for ARM software and desktop (x86) software](image)

Figure 4: Tools availability for ARM software and desktop (x86) software
3. Application and Driver has dependency on System Software. Because ISS purely simulates a target processor, complete embedded software stack including system software is required to be built before being able to run on VP. This requirement forces application and driver developers to wait for the arrival of the system software. In addition to the delay of running on VP, early stage bugs in system software tend to introduce significant destructions, making it harder to identify bugs in applications and drivers.

These challenges directly degrade the embedded software development productivity that is one of the significant success factors in today’s SoC design flow. In this work, we address these issues by proposing a new portable abstraction framework for VP which:

1. Removes ISS from VP by directly executing production software stacks on host, achieving native software execution time on host

2. Activates rich suites of desktop development tools in host by compiling production software stacks in host binary

3. Allows system software independent application and driver development

The trade-off of using our framework is that processor architecture details such as processor registers, cache and pipeline will be lost due to absence of ISS. However, our work focuses on application and driver development which do not require processor architecture details extensively.
1.2 Contributions and Benefits

Contribution of this work is as follows:

- Demonstrating that ISS-less VP methodology is a promising candidate for early embedded software development

While achieving above contribution, this work also presents the following benefits which directly improve the engineering productivity.

- Removing dependency on system software in application and driver development
- Removing system software bugs
- Saving significant overhead of developing and debugging ISS for a new processor
- Achieving desktop software development environment for embedded software
- Demonstrating that the ISS-less VP can achieve performance improvement up to 2000% over conventional VP with state-of-the-art ISS performance

1.3 Thesis Overview

Chapter 2 reviews background information in the area of SoC design with focus on software development with VP. The section ends with brief explanation of QEMU, a state-of-the-art open-source VP and related work. Chapter 3 focuses on the abstraction framework for ISS-less VP and ends with a complete example of a network application running on a real-world OS on the framework. Chapter 4 discusses our custom software stack to achieve system software independent application and driver development on VP. The chapter also provides a complete case study with H.264 video decoder running on the software stack. Chapter 5 provides
evaluation and analysis of the framework in terms of simulation performance, portability and software productivity. Lastly, chapter 6 presents summary and contributions with future work suggestions.
Chapter 2
Background

2 Background

2.1 SoC design flow

A typical traditional SoC design flow shown in top diagram of Figure 5 started with hardware development consists of algorithm design, architecture design, IP development, chip integration, verification and post-silicon bring-up with possible re-spins. Once the entire hardware cycle is finished and evaluation board has arrived, software teams begin their software profiling and development on the board. In recent years, due to exponential growth in SoC complexity and ever demanding software features by the user, time-to-market for software has grown to become the bottleneck in the entire SoC design flow. Market research firm International Business Strategies Inc claims that with 90nm, software development effort has already surpassed hardware development and is projected to dominate even more of the SoC design time [16]. Today with 45 nm technologies, only less than 40% of time is spent on hardware [16]. With this trend, industry has invalidated the traditional SoC design flow and adapted to simulation technologies to enable hardware/software co-development. This enabled early software development before the hardware evaluation board arrives with simulation-based virtual hardware. Various simulation technologies exist ranging from cycle-accurate RTL simulations, In-Circuit Emulation (ICE) to VP. All three approaches have their advantages and disadvantages.
Cycle-accurate RTL simulations model very accurate hardware behaviors down to register transfer level with system clock, suitable for hardware verification and profiling. Examples of the simulators are Mentor ModelSim [34], Synopsys VCS [35] and Cadence Incisive Enterprise Simulator [36]. This simulation methodology provides very accurate timing with sacrifice of simulation time. The performance achieved is usually in the range of single-digit hertz for large designs [31] which is not even close to being able to run reasonable amount of software code on it.
On the other hand, ICE emulates the entire SoC hardware by synthesizing and running SoC RTL on Field Programmable Gate Array (FPGA). This provides reasonably fast execution time for software development. Today, a number of companies provide complete solutions for ICE such as Veloce from Mentor Graphics [42], ICE from Lauterbach [44] and ZeBu from EVE [43]. However, it is ready only after the RTL has been completed and is often too late to impact any architectural and performance issues in the current design.

Full-system VP is fully functional software models of SoC hardware. It usually consists of purely functional ISS and bus/peripheral models. The models are implemented with focus on functionality rather than hardware implementation details to achieve faster simulation speed. A typical VP is able to simulate Linux booting in tens of seconds making it fast enough for complex software development. Because it only requires functional specification of hardware, VP can be constructed after algorithm phase of design cycle as shown in bottom diagram in Figure 5. This allows software development to begin much earlier in the project timeline. Examples of VP are Synopsys Innovator [37], Wind River Simics [38], QEMU [3] and Bochs [39]. Our work focuses on embedded software and VP framework and directly contributes to the technologies. Table 1 summarizes properties of the three simulation methodologies described in this section.
<table>
<thead>
<tr>
<th>Feature</th>
<th>RTL</th>
<th>ICE</th>
<th>VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Speed</td>
<td>Low</td>
<td>High</td>
<td>Very High</td>
</tr>
<tr>
<td>Availability</td>
<td>Middle (IP stage)</td>
<td>Late</td>
<td>Early</td>
</tr>
<tr>
<td>Development Time</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Visibility in platform internals</td>
<td>Limited</td>
<td>Very Low</td>
<td>High</td>
</tr>
<tr>
<td>Setup and execution time</td>
<td>Medium</td>
<td>High (including board setup)</td>
<td>Low</td>
</tr>
</tbody>
</table>

Table 1: Properties of different simulation methodologies

2.2 Virtual Prototype

A Virtual Prototype (VP) is a set of fully functional software models of hardware that together form a single or multicore SoC. VP runs on a host often a general-purpose PC and leverages ISS to run unmodified production embedded software including applications, OS, firmware, drivers and bootloader. The simulation speed depends on how detailed ISS and hardware models are. However for software development, cycle-accuracy is not an important factor because often software 1) does not care about hardware implementation but its functionalities, 2) does not need to know the exact timing behavior, and 3) only directly interacts with hardware interfaces (i.e. registers and memory-mapped bus). Hence, hardware models in VP tend to implement only software-visible functionality and interfaces, resulting in much simpler models.

Figure 6 shows a VP for a typical wireless SoC platform. White boxes represent software models of bus and peripherals while shaded boxes represent ISS for two processors (ARM and DSP). The models are often implemented using high-level languages such as SystemC [6], VRE
C++ and pure C/C++. Communications between models are achieved by calling interface functions of destination model with emphasis on functionality of the data transfers (what data are transferred to and from what locations) instead of its actual implementation. The simulation is controlled by a simulation kernel which schedules the execution of the models to emulate the real SoC behavior. VP offers number of benefits over other hardware-based simulation methodologies (RTL simulation and ICE):

1) *Early availability before RTL completion for early-software development*: Because the software models implement only functionality of the hardware, the models can be designed before the hardware implementations are figured out. This leads to a very early availability of VP allowing software team to start testing in early stage of the project. Figure 5 also shows how SoC design flow using VP can allow software design to begin just after hardware algorithm stage.

2) *Supreme visibility of both software and hardware states*: Because VP is fully implemented in software models controlled by a simulation kernel, it is trivial to observe software and hardware states such as registers and interrupt lines at any given time.

3) *Simultaneous breakpoint for the entire SoC*: Simultaneous breakpoint on the entire SoC is critical in multi-processor SoC to observe software and hardware states at a given time. In ICE, because the entire SoC is running on FPGA, simultaneous breakpoint on the entire SoC is next to impossible due to interrupt delays and lack of breakpoint facilities in hardware. However, in VP, it is trivial because the simulation is controlled by a simulation kernel running on a sequential host CPU. By simply pausing the simulation scheduler automatically halts the entire SoC achieving the simultaneous breakpoint.
4) Tens of Million Instructions Per Second (MIPS) simulation speed: Because of the high abstraction of hardware details, typical VP can achieve tens of MIPS [32][33]. The high simulation performance allows software developers to boot complex software stacks such as Embedded Linux in tens of seconds, making it suitable for today’s complex embedded software development.

![Diagram](image)

**Figure 6: A Virtual Prototype example**

Wide varieties of VP implementation exist in industry and most of them are designed in-house to suite the company’s SoC development needs. Historically, several companies such as Virtio, CoWare and VaST (now Synopsys) have introduced VP framework using proprietary
technology components. Over the last few years, with growing importance of interoperability between the VP models, Open SystemC Initiative’s (OSCI’s) Transaction-Level-Modeling (TLM) Working Group has released the TLM-2 standard, which includes the transaction abstractions for hardware models. Today, more and more companies are adapting their VP models to the TLM standard for better portability. Now there are growing ecosystems based on the TLM using SystemC such as AMBA bus, IBM, MIPS and ARM processor models offered by Synopsys’s DesignWare System Level Library [5]. There are also number of commercial tools which provide bundle of GUI, simulation kernel and model libraries for VP such as Synopsys Innovator [37] and Wind River Simics [38].

There also exists open-source VP such as Quick Emulator (QEMU [3]) and Bochs [39]. Both VP implement their own simulation kernel as well as various common ISS and hardware models contributed by Open Source community. In this work, we have chosen QEMU as our reference VP and performance reference due to its: 1) state-of-art performance, capable of achieving simulation speed of hundreds of MIPS [17], 2) wide variety of supported hardware models and ISS, 3) framework for models to communicate with real world (i.e. network packets and LCD screen) and 4) proven popularity in academia [18] and industries [40].
2.3 QEMU

QEMU is an open source VP which implements dynamic translation based ISS that runs unmodified software image compiled for a target processor. It has been ported to many popular target processors such as ARM, MIPS, PPC, Microblaze, Nios and x86. With its efficient implementation of ISS, it is able to achieve hundreds of MIPS simulation performance [17]. QEMU also defines a set of APIs for implementing bus and peripheral models for building an entire SoC platform. Open source community has already implemented many popular platforms on QEMU such as ARM integrator board and Texas Instrument’s OMAP platform. In addition to the above features, QEMU presents set of APIs to connect simulation host’s peripherals to the emulated peripheral models to interact with outside world such as keyboard input, LCD display and Ethernet packets.

Figure 7 shows an ARM platform built using QEMU. It contains ARM ISS which dynamically translates embedded software in ARM binary to host instruction sets and executes on the host. Peripherals are connected to the ARM ISS using a memory-mapped bus and can be triggered by memory-mapped I/O access to their registered memory regions. QEMU also allows peripherals to interact with real peripherals on simulation host to talk to outside world. Table 2 shows several examples of connecting peripheral models with host peripherals. For reasons described, QEMU has proved to be a fast VP candidate and is widely used in academic researches [18] and industry such as Android Emulator [40]. In this work, we use QEMU as our base VP as well as performance reference. In the following sub-sections, we describe relevant internals of QEMU.
Figure 7: ARM SoC platform built using QEMU

<table>
<thead>
<tr>
<th>Host Peripheral</th>
<th>SoC Peripheral Model</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal</td>
<td>UART</td>
<td>User input and standard output</td>
</tr>
<tr>
<td>LCD</td>
<td>LCD</td>
<td>Display emulated frame buffer to host LCD</td>
</tr>
<tr>
<td>Ethernet</td>
<td>Ethernet</td>
<td>Send and receive Ethernet packets using host networking</td>
</tr>
<tr>
<td>Timer/RTC</td>
<td>Timer/RTC</td>
<td>Get current time from host Real-Time Clock (RTC) or timer</td>
</tr>
<tr>
<td>Audio</td>
<td>Audio</td>
<td>Output sound to host speaker or input via microphones</td>
</tr>
</tbody>
</table>

Table 2: Example of mapping SoC peripheral models with host peripherals in QEMU
2.3.1 Dynamic Translation Based ISS

A number of implementations exist for ISS with major ones being Interpretation and Dynamic Translation. Interpretation is a simple approach where every instruction compiled for the target processor is fetched from simulated memory, decoded and then emulated the equivalent behavior with a bunch of host instructions. Cons of this approach are that there are many redundant decoding per each target instruction especially in loop and large overhead of memory accesses to fetch and store simulated registers (simRegs in Figure 8). The instruction execution time in Interpretation-based ISS tends to be slower than native execution on simulation host by magnitude of 3 [29]. QEMU uses Dynamic Translation which is more complex but much more efficient. Instead of decoding instructions one by one, it fetches an entire basic block and translates it into a Translation Block (TB) in host binary. The TBs are then stored in a cache called Translation Cache for future reuse in order to avoid redundant decoding of the same instructions. Translated instructions are then executed directly on host memory to achieve near native execution. The performance of dynamic translation improves significantly over interpretation and can achieve speeds up to 50% of native execution on host [41].
2.3.2 Peripheral Model / Host Peripheral Access

QEMU also provides framework to map peripheral models to host peripherals. Figure 9 shows a simplified picture of peripheral model access and host peripheral interaction in QEMU using UART as an example. When executing a TB on simulation host, any memory-mapped I/O access is trapped and invokes a register access in a peripheral model corresponding to the memory address. For UART model, it may simply fetch data from FIFO and return to ISS. However, there needs to be a mechanism to update UART FIFO with actual inputs from user through terminal in simulation host. QEMU implements this by a means of signal handler associated with OS signals (i.e. SIGIO and SIGALRM in Linux). ISS is interrupted every configurable interval (typically 1 ms) using host SIGALRM or at every occurrence of an I/O event using SIGIO. The interruption causes a signal handler to be triggered. Inside the handler, host peripherals are
selected one by one to see if any events have occurred resulting in a character typed in host terminal to be caught. The character is then sent to UART peripheral model to update its FIFO and possibly raising an interrupt. Hence, next time ISS accesses UART registers, it will see the illusion of a character arrival at the UART peripheral.

Figure 9: Peripheral Model Access and host peripheral interaction in QEMU
2.4 Previous Work

Open SystemC Initiative (OSCI) has released TLM-2 standards [6] for VP models with different timing models such as Untimed, Cycle-approximate and Cycle-accurate models. The standard mainly defines hardware and software component interactions and it is up to the user to implement software component using ISS. Our work focuses on abstraction layer inside the software itself to enable ISS-less VP.

Monton et al. [18] have proposed an emulation framework to mix QEMU with SystemC models. In the work, QEMU is basically acting as an ISS driving the SystemC models of peripherals targeting driver and software development. Our approach focuses on the software itself to remove ISS from QEMU.

In recent years, a significant amount of work has been focusing on speeding up virtual prototype simulation while preserving reasonable timing accuracy. A quick summary of related work is described in Table 3.

A. Bouchhima et al. [10] proposed methodology to execute application software natively while abstracting away OS details using an OS API. OS Model Library was provided to support implementing OS API as well. M. Krause et al. [11] have presented a framework to abstract away RTOS model from original cross-compiled production software and achieve native execution of RTOS model while executing application on ISS. Although both approaches achieve significant speed-up, OS model is required instead of a real OS and only part of the entire production software stacks is executed natively on host. In comparison, our approach is able to execute the entire software stacks natively with real OS.

S. Yoo et al [7] presented a fast and accurate software simulation model by enabling native execution of application and OS. Synchronization function and an adaptation layer are
introduced to enable synchronization between software/hardware models and processor-dependent code abstraction respectively. A similar framework is presented by P. Gerin et al. [12] by defining hardware/software interface. Ecker et al. [13] have presented a dataflow abstracted VP for driver development and achieved ISS-less simulator from SystemC-based VP. All three works achieved a magnificent simulation speedup but required embedded software to be implemented as SystemC process. Not enough benchmarks or real-life applications were used in experiments as well. Our approach allows embedded software to execute as a standalone process independent of SystemC models. In addition, our framework supports system software independent application and driver development.

On top of speeding up the simulation, number of work has focused on generating software with embedded timing information. In [8] and [9], embedded software code is analyzed at basic block level and is re-compiled into a delay-annotated code. Both works demonstrated the generated software running natively on simulation host using SystemC and achieved reasonable accuracy of software timing. Our work focuses more on fast and component software development methodology for “non-real-time” software components rather than software timing accuracy required by real-time software.
<table>
<thead>
<tr>
<th></th>
<th>ISS</th>
<th>Compiled SW</th>
<th>Native Execution</th>
<th>RTOS</th>
<th>Software needs to be built</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>NO</td>
<td>SystemC</td>
<td>Application</td>
<td>Model</td>
<td>Application + RTOS Model + Drivers</td>
</tr>
<tr>
<td>11</td>
<td>Application</td>
<td>Target Binary</td>
<td>RTOS</td>
<td>Model</td>
<td>Application + RTOS Model + Drivers</td>
</tr>
<tr>
<td>7</td>
<td>NO</td>
<td>SystemC</td>
<td>Entire</td>
<td>Real</td>
<td>Application + RTOS + Drivers</td>
</tr>
<tr>
<td>9</td>
<td>NO</td>
<td>SystemC</td>
<td>Entire</td>
<td>NC</td>
<td>Application + RTOS + Drivers</td>
</tr>
<tr>
<td>8</td>
<td>YES</td>
<td>SystemC</td>
<td>NC</td>
<td>NC</td>
<td>Application + RTOS + Drivers</td>
</tr>
<tr>
<td>13</td>
<td>NO</td>
<td>SystemC</td>
<td>Entire</td>
<td>Model</td>
<td>Application + RTOS Model + Drivers</td>
</tr>
<tr>
<td>12</td>
<td>NO</td>
<td>SystemC</td>
<td>Entire</td>
<td>Real</td>
<td>Application + RTOS + Drivers</td>
</tr>
<tr>
<td>Ours</td>
<td>NO</td>
<td>Executable</td>
<td>Entire</td>
<td>Real</td>
<td>Application + Drivers</td>
</tr>
</tbody>
</table>

Note: NC = Not Clearly Described

Table 3: Summary of related work on speeding up virtual prototype

2.5 Summary

This chapter reviewed various background topics in SoC design flow with focus on software and driver development. It first discussed traditional SoC design flow, problems and today’s new hardware/software co-development flow. It then focused into a common solution for early software development, VP and its variations. QEMU was explained as an open VP with outstanding simulation performance and wide acceptance. Our work uses QEMU as our base VP and performance references. In order to provide comparison with our work, related work in the VP field was reviewed after.
Chapter 3
ISS-less Virtual Prototype Framework

3 ISS-less Virtual Prototype Framework

3.1 Objectives

The framework presented in this chapter:

- Removes ISS from VP by directly executing embedded software stack in host, achieving native software execution on host
- Activates rich suites of desktop development tools in host by compiling embedded software stack in host binary

3.2 Key Ideas

The main idea of the framework is to build embedded software designed for target in host binary by abstracting away components that are specific to target architecture and platform. As it turns out, embedded software has very little dependency on instruction set architecture itself but rather on computer organization such as peripheral memory map and interrupt delivery mechanism. Figure 10 illustrates the independence of instruction set from majority of embedded software. When software is written in high-level language such as C, a compiler for target processor is used to convert C program to target instruction sets. Compiler essentially provides abstraction of underlying instruction set architecture and allow programmer to write same code for different processors. In contrast, Figure 11 shows dependency on memory-map in embedded software. A memory-mapped I/O access to certain peripheral may not work on different SoC platform because the peripheral mapping may be different. In this case, software (typically driver) needs to be modified to access corresponding memory-mapped I/O address.
Because of the independence on instruction set architecture, by properly abstracting away the computer organization dependent software codes, it is possible to compile embedded software into host instruction set (i.e. x86) and execute directly on host processor. Naturally, conventional ISS is no longer required from VP. In addition, software testing can be carried out directly on host binary as a host executable, activating all rich suites of desktop development tools.

After investigating many existing embedded software stacks ranging from Embedded Linux to standalone bare-metal firmware, we have concluded that computer organization dependent codes can be categorized into three, Memory-Mapped I/O Access, Interrupt/Exception Delivery and Processor Architecture. Examples of each component are shown in Table 4.
Memory-mapped I/O includes any driver calls which access peripheral registers directly (with or without Memory Management Unit (MMU)). Peripheral access needs to know the exact memory-mapped address visible from target processor which can vary on different SoC platforms. Interrupt/Exception delivery concerns the target processor specific methodology for handling interrupts and exceptions. For example, ARM processor implements seven-entry interrupt vector table for each of interrupt/exception type as shown in Figure 12 at a fixed location in memory. Each table entry contains a jump instruction to its handler. Therefore, when an IRQ occurs, the current Program Counter (PC) will jump to IRQ vector table location and will immediately execute the handler for IRQ. Finally, Processor Architecture includes all other processor dependent components such as jump buffer structure for context switching and stack pointer setup.

| 0x0 | Reset       |
| 0x4 | Undefined Instruction |
| 0x8 | Software Interrupt   |
| 0xC | Prefetch Abort       |
| 0x10 | Data Abort         |
| 0x18 | IRQ               |
| 0x1C | FIQ               |

Figure 12: ARM interrupt vector table

By introducing abstraction API for codes in these categories, the software stack will be completely free of dependency from underlying hardware and can be compiled and executed on any computer environment including the host.
<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory-Mapped I/O Access</td>
</tr>
<tr>
<td>Interrupt/Exception Delivery</td>
</tr>
<tr>
<td>Processor Architecture</td>
</tr>
</tbody>
</table>

Table 4: Architecture and Platform dependent components in embedded software

### 3.3 Framework Architecture

Figure 13 presents the high-level architecture of our framework. It consists of at least three separate processes, Software Stack, ISS-less VP and Broker. Software Stack process consists of the entire embedded software stack built on top of three abstraction APIs, I/O Memory API, Interrupt API and Arch. Each API abstracts away the three categories of computer organization dependencies described in Table 4 respectively. The APIs in turn implement communication mechanism with VP to preserve hardware interaction. After all computer organization dependent codes are written on the APIs, embedded software can be compiled into host binary and executed natively on host processor. VP process now contains no ISS since the software stack is executed directly on the host processor. It simply serves as collection of peripheral models communicating with software stack process via the abstraction APIs. Since SoC often contains multiple processors such as ARM and DSP resulting in multiple embedded software stacks and ISS, the communications between software stacks and VP are implemented using a broker to enable point to point access between multiple processes.
3.3.1 I/O Memory API

I/O Memory API is used by device drivers in embedded software to trap memory-mapped I/O accesses. The API provides functions for read, write and initialization shown in Figure 14.
At initialization, embedded software needs to call `iomem_init()` to register the software stack process with the broker to obtain a unique address. Once the connection and registration are complete, drivers can read and write on memory-mapped I/O by calling `iomem_read()` and `iomem_write()` functions respectively. I/O Memory API will encode the memory-mapped I/O access to our custom message protocol and send out to the Broker destined to corresponding VP process.

Figure 15 shows how using I/O Memory API enables drivers to access desired peripheral model in VP when running natively on host. For example, in a VP designed for an ARM based SoC, a UART model is accessed by memory-mapped I/O at 0x10000. When ARM ISS accesses the address, a request is sent to the UART model and appropriate hardware emulation is executed (Figure 15 a). However, once we compile the same software stack in host binary and run directly

```c
/* Connect and register software process with Broker */

int iomem_init()

/* Read a Memory-mapped I/O at address base for size Bytes and save to buf*/

void iomem_read (unsigned int base, void *buf, int size)

/* Write buf to Memory-mapped I/O at address base for size Bytes */

void iomem_write (unsigned int base, void *buf, int size)
```

Figure 14: I/O Memory API
on x86 host, the memory access to 0x10000 will likely result in segmentation fault because the particular UART peripheral does not exist at 0x10000 (Figure 15 b). Now, consider adding an I/O Memory API at memory-mapped I/O access. The memory access will be trapped by the API which implements message encoding and communication to VP process via a broker. Once the encoded message is arrived at VP, it is decoded and the UART model executes the correct emulation (Figure 15 c). While achieving the exact same functionality as Figure 15 a, the advantage here is that it completely removes the reliance on the actual memory access to drive a peripheral model, resulting in memory-map independent I/O.

Another key note is that embedded software built on the I/O Memory API also can be directly compiled on target instruction set to be used for production. We enable the full-portability by providing a conditional flag to switch the API implementation to direct memory access.

![Figure 15: Memory-mapped I/O abstraction illustration](image-url)
3.3.2 Interrupt API

While Memory-Mapped I/O accesses are processor-initiated requests (aka. downcall), there exits peripheral initiated request (aka. upcall), namely Interrupt. It is a mechanism for peripherals to asynchronously modify the current PC in processor and force the global Interrupt Service Routine (ISR) to be called. The global ISR then looks at which peripheral has raised the interrupt and call corresponding peripheral interrupt handler to serve the request. Different processor architecture have slightly different ISR triggering mechanism. However in general, they reserve a section of memory for a vector table which stores the address to the global ISR. For example in ARM architecture, external interrupt from a peripheral will set PC to 0x18 where an instruction to jump to global ISR is stored (Figure 16). In VP, this mechanism is usually implemented in ISS, hence is not useful in our framework as its objective is to discard ISS from VP itself.

![Figure 16: Conventional VP interrupt delivery mechanism](image-url)
Our framework implements a solution to interrupt delivery using OS signal facility on host. The idea is to send an inter process signal from VP process directly to the software stack process where global ISR is registered as the signal handler for a particular signal. This way, the software process can be interrupted asynchronously and execute the global ISR as if caused by a real peripheral. After the ISR completes the service, the embedded software will resume its execution from where it was interrupted, which is consistent with the behavior of real external interrupt. Figure 18 shows a single API we provide for setting up this interrupt mechanism. Figure 17 illustrates the interrupt delivery flow in our framework using UART as an example:

1. At start-up, the software stack calls `interrupt_register()` to notify the VP process with its process ID (100). The API also registers global ISR, (*handler) with a particular OS signal.

2. At later time, the UART model notifies the interrupt controller model to raise an interrupt due to a data arrival.

3. The interrupt controller model uses Interrupt API to raise the OS signal directly to the software stack process using process ID of 100.
(4) The software stack process receives the signal and asynchronously switch current context to the global ISR which was registered as signal handler for the arriving signal. The global ISR analyzes the interrupt source and runs the UART interrupt handler to serve the request.

```c
/* Register Process ID of software process with VP */

int interrupt_register(pid_t pid, void (*handler)(int))
```

Figure 18: Interrupt API

3.3.3 Arch

Arch is another abstraction API for all other target processor architecture specific codes in a software stack. Examples are shown in Table 5. In order to run a software stack on a host, Arch components need to be abstracted and linked with corresponding implementation for host architecture. For Embedded Linux, we have already identified the functions need to be replaced and are also shown in Table 5. Our framework already offers Arch package for x86 host. While replacing Arch components with host implementation still holds the correct functionality, it is one of the limitations of our framework that the Arch implementation is not consistent with the final embedded software. However, these architecture-dependent codes usually have been well tested by processor vendors and often are not significant concerns for application, device driver and system software development.
### Table 5: Non-I/O architecture dependent components in Software Stack

<table>
<thead>
<tr>
<th>Category</th>
<th>Where to find in Embedded Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump buffer</td>
<td>setjmp(), longjmp()</td>
</tr>
<tr>
<td>Heap structure</td>
<td>sbrk()</td>
</tr>
<tr>
<td>Processor Interrupt Enable/Disable</td>
<td>Initialization assembly code</td>
</tr>
<tr>
<td>Stack and interrupt vector table setup</td>
<td>crt0.S</td>
</tr>
<tr>
<td>Processor endian definition</td>
<td>endian.h</td>
</tr>
<tr>
<td>Linker Script</td>
<td>board.lds</td>
</tr>
<tr>
<td>Any assembly codes</td>
<td>Any where found</td>
</tr>
</tbody>
</table>

3.4 Network Software Stack Example with RTOS

We will show a complete example of a software stack built on a real-life OS called Contiki [19] running a network application on our ISS-less VP framework. Contiki is an open source, highly portable, multi-tasking operating system targeting mainly for networked embedded systems and wireless sensor networks. The beauty of it is in small memory footprint which is about 2 kB of RAM and 40 kB of ROM in its typical configuration. We will be using an ARM9 based custom SoC consisting of number of different peripherals taken from different boards for demonstration throughout this work. Figure 19 shows the SoC diagram and the origins of each peripheral component and their drivers.
### Table 3.1: Device Category and Model Details

<table>
<thead>
<tr>
<th>Category</th>
<th>Model</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISS</td>
<td>ARM9</td>
<td>N/A</td>
</tr>
<tr>
<td>UART</td>
<td>USRP2 [20] UART core</td>
<td>USRP2 firmware suite</td>
</tr>
<tr>
<td>LCD</td>
<td>LCD from S3C2410 [21] board</td>
<td>U-Boot S3C2410 Port</td>
</tr>
<tr>
<td>PIC</td>
<td>Xilinx Interrupt Controller</td>
<td>Xilinx EDK driver [22]</td>
</tr>
<tr>
<td>Ethernet</td>
<td>Xilinx Ethernet Lite</td>
<td>Xilinx EDK driver</td>
</tr>
<tr>
<td>Frame Buffer</td>
<td>Section of DRAM</td>
<td>N/A</td>
</tr>
<tr>
<td>Timer</td>
<td>Custom Timer Core</td>
<td>Custom</td>
</tr>
</tbody>
</table>

**Figure 19:** SoC platform built in QEMU for demonstration

### 3.4.1 Conventional VP flow

Before we demonstrate our framework, we first show how conventional VP emulates the behavior of networking. We have first created the reference SoC VP described in Figure 19 with ISS using QEMU. Contiki was then ported to ARM architecture and included device drivers for the peripherals. At last, we have compiled Contiki with TCP/IP stack enabled using ARM cross-compiler and successfully demonstrated that a packet from another computer to QEMU platform
has reached TCP/IP stack in Contiki and sent the response back to the packet sender. Figure 20 shows life time of a ping request from a packet sender to Contiki on QEMU and returned.

(1) A computer sends a packet to the host addressing the IP address of Contiki

(2) The packet is received on the host Ethernet device and is delivered to the Ethernet peripheral model in QEMU

(3) The Ethernet model saves the packet in its FIFO and instructs Programmable Interrupt Controller (PIC) to raise an interrupt to notify the packet arrival

(4) PIC accepts the request and raises an external interrupt to ARM ISS

(5) ARM ISS switches its mode to Interrupt Mode and adjusts the PC to Global ISR via Interrupt Vector Table.

(6) Global ISR reads PIC register via PIC driver to find out that the Ethernet model was the source of interrupt and invokes the receive handler in TCP/IP stack

(7) TCP/IP stack then reads the actual packet via the Ethernet driver and generates response

(8) The response packet is then sent out to Ethernet peripheral model

(9) Ethernet peripheral model sends the packet to the host Ethernet device and finally reaches back to the packet sender

This example exercises multiple drivers (PIC and Ethernet), Interrupt delivery to request service for Ethernet packet arrival. In next few sub-sections, we will discuss our experiences of porting the Contiki and QEMU to our framework and present a Software Stack process running Contiki OS natively on host.
3.4.2 I/O Memory API Porting

First task was to replace Memory-Mapped I/O Accesses found in drivers with our I/O Memory API. This was trivial task as most well written drivers have macro in form of Hardware Abstraction Layer (HAL) which implements the lowest level function to access memory-mapped I/O. For example, Xilinx’s Ethernet and PIC drivers had macros for accessing 8/16/32 bit memory and all we needed to do was to replace the implementation with our I/O Memory API shown in Figure 21. After, all memory-mapped I/O has been ported to I/O Memory API, we simply added function call to `iomem_init()` in Contiki’s start-up code.

Figure 20: Life time of a packet on Contiki with QEMU
```c
#define IO_In32(addr) *((uint32*)addr)

uint32 driver_read(addr) {
    return IO_In32(addr);
}
```

```c
#define IO_In32(addr) \\ do{ uint32 buf; \\    iomem_read(addr, &buf, 4); \\    return buf; \\ } while(0);
```

```c
uint32 driver_read(addr) {
    return IO_In32(addr);
}
```

```c
start:
    iomem_init();
```

Figure 21: Example of I/O Memory API porting in a real-life driver. Left: original driver and Right: ported to I/O Memory API

### 3.4.3 Interrupt API Porting

Interrupt API porting is shown in Figure 22. We first provided a wrapper function for global ISR function found in PIC driver. The wrapper was simply needed to conform to the host signal handler format (i.e. void (*func) (int)). Finally we added call to `interrupt_register()` to register the Process ID of Contiki with QEMU at start-up of the software stack.
void global_ISR(void) {
    /* check source of intr */
    /* execute matching intr handler */
    /* acknowledge intr */
}

void global_ISR(void) {
    /* check source of intr */
    /* execute matching intr handler */
    /* acknowledge intr */
}

void global_ISR_wrapper(int sig) {
    global_ISR();
}

start:
    pid = getpid() /* get my process ID */
    interrupt_register(pid, global_ISR_wrapper);

Figure 22: Example of Interrupt API porting in a real-life embedded software stack. Left: original global ISR. Right: ported to interrupt API

3.4.4 Arch Porting

Arch porting required some work to identify the locations of the Arch components in Contiki. Table 6 shows the major actions taken in Arch porting.

- Replaced setjmp and longjmp implemented in ARM assembly with host implementation.
- Replaced sbrk() call to dynamically allocate heap memory with host heap memory range.
- ARM assembly code for enabling and disabling global interrupt was simply removed because the Interrupt API serves the same functionality.
- Stack and interrupt vector table setup were replaced by host crt0.s file which serves the same purpose.
- Processor endianness happened to be same between ARM and x86 host (little endian), however we have included host endian header file (i.e. endian.h) into Contiki for consistency.
- Linker script was replaced by default linker script found in host compiler to map the embedded software onto host memory.
- No other assembly code was found in ARM port of Contiki.

<table>
<thead>
<tr>
<th>Category</th>
<th>Actions Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump buffer</td>
<td>Replaced with host setjmp/longjmp()</td>
</tr>
<tr>
<td>Heap structure</td>
<td>Replaced with host sbrk()</td>
</tr>
<tr>
<td>Processor Interrupt Enable/Disable</td>
<td>Removed</td>
</tr>
<tr>
<td>Stack and interrupt vector table setup</td>
<td>Replaced with host crt0.s</td>
</tr>
<tr>
<td>Processor endian definition</td>
<td>Includes endian.h in host GCC</td>
</tr>
<tr>
<td>Linker Script</td>
<td>Replaced with host linker script</td>
</tr>
<tr>
<td>Any assembly codes</td>
<td>Not Found</td>
</tr>
</tbody>
</table>

Table 6: Arch porting for Contiki
3.4.5 Compiling Contiki

After Contiki was successfully ported on the abstraction APIs, we have compiled Contiki with host compiler (instead of ARM cross-compiler) and generated a x86 desktop executable.

3.4.6 Modifications to QEMU

Finally, a few modifications to QEMU were performed to adapt to our ISS-less VP framework. This included removing ISS, implementing a simple lookup table for incoming I/O Memory API and replacing QEMU’s ISS interrupt mechanism with our Interrupt API.

Top diagram in Figure 23 illustrates a simplified version of QEMU implementation. The ISS infinite loop dynamically translates target instructions to host instructions and directly runs on host processor. When a memory-mapped I/O instruction is encountered, it triggers corresponding peripheral model execution. When a peripheral model needs to raise an interrupt, QEMU informs ISS to simulate interrupt behavior by adjusting program counter to the global ISR location. The next target instruction will preempt the current basic block and execute global ISR function to serve the interrupt.

To adapt QEMU to our framework, we have first removed ISS infinite loop function completely as our framework does not require ISS. Second, we have replaced the loop by non-blocking polling for any incoming IO Memory API messages from the Software Stack process. Upon a message arrival, the message is extracted into memory address and data. The address is used as an index to a look-up table for choosing corresponding peripheral model. Lastly, QEMU’s interrupt mechanism was replaced with our interrupt API to directly send signal to software stack process to execute global ISR (bottom figure of Figure 23).
Figure 23: Modifications to QEMU. Top: Original QEMU, Bottom: QEMU with ISS-less VP framework
3.4.7 Contiki execution flow on our framework

After all steps described in section 3.4 have been completed, we dispatched three processes, namely software stacks (Contiki x86 executable), ISS-less VP (QEMU) and Broker, directly on the host. A network packet was successfully received by Contiki and responded to the packet sender. The behavior was consistent with our demonstration with conventional VP with ISS described in 3.4.1 while successfully removing ISS and achieving native software execution of Contiki. Figure 24 shows the new life time of a network packet traveling on our framework.

1. A computer sends a ping packet to the simulation host addressing the IP address of Contiki
2. The packet is received on host Ethernet device and is delivered to Ethernet peripheral model in QEMU
3. The Ethernet model saves the packet in its FIFO and instructs PIC to raise an interrupt to notify the packet arrival
4. PIC accepts the request and raises a signal on host to the process ID of Contiki process via Interrupt API
5. Global ISR which is registered as the signal handler via Interrupt API is asynchronously called
6. Global ISR then reads PIC register via PIC driver to find out that Ethernet was the source of interrupt and invokes packet receive handler in TCP/IP stack
7. TCP/IP stack then reads the actual packet via the Ethernet driver and generate response to the packet
8. The response packet is then sent out to Ethernet peripheral model via I/O Memory API
(9) After traveling through Broker, the response packet is arrived at Ethernet model in QEMU and sent back to the packet sender through host Ethernet device.

Figure 24: Life time of network packet with ISS-less VP Framework

3.5 Summary

This chapter discussed key ideas of our framework to achieve ISS-less VP and native execution of embedded software stacks on a host. The high-level framework architecture and abstraction APIs were first described followed by detailed implementation of each API. The chapter ended with a complete porting example and demonstration of a real-life OS, Contiki, and open source VP, QEMU running on our framework and achieving a functionality of a TCP/IP server.
Chapter 4
System Software Independent Application and Driver Development

4 System Software Independent Application and Driver Development

4.1 Objectives

To test software on VP, the entire software stacks with system software need to be built. However, it is non-trivial to build the entire production software stack as it requires porting of OS (e.g. Embedded Linux), firmware and bootloader. This is often time consuming and error-prone process even for system software engineers resulting significant wait time for application and driver developers before testing their software components on VP. In addition, system software bugs tend to introduce productivity destruction to application and driver developers making it harder to spot their own bugs.

In this section, we present a highly portable bare-metal software stack which is readily be used by application and device driver developers to debug their code with similar environment as the production embedded software. It contains well-tested minimal set of system software components to reduce bugs outside of developer’s interest. The software stack consists of:

- A simple multi-tasking operating system
- POSIX-compliant standard libraries
- Well-defined device driver framework
- TCP/IP stack for basic network functionality
- Virtual File System (FS) built on either Network FS or host FS
• Adaptation to ISS-less VP framework

With this software stack, application and device driver developers can independently debug their codes immediately on VP. The software stack also provides a well abstracted environment from other components of the software stack (i.e. system software) for a clean debugging experience. Moreover, once the system software is delivered, POSIX-compliant applications developed using our software stack is directly portable to the production software stack and device drivers written in our driver framework can be easily ported by applying an appropriate driver wrapper for corresponding operating system.

Our software stack supports two usage models for application and driver development. Usage Model I is to divide our software stack into two parts, Application process and Emulated Kernel process to run application and drivers in separate processes (Figure 25). This usage model is suitable for early stage of development as application developer is able to start developing its functionality without driver existence and drivers can also be developed as a standalone process directly on VP. The interactions between the two processes can be controlled by user as development progresses. Usage Model II is to combine the two processes into a standalone software stack becoming similar to the production software stack (Figure 26). This usage model eliminates communications between Application and Emulated Kernel Processes resulting in better simulation performance. The model is recommended for later stage of development when both applications and drivers are reasonably stable. When using same application and drivers, both usage models achieve the exactly same behavior while creating different level of abstraction for application and driver developers.
Figure 25: Usage Model I: Separate processes for applications and drivers

Figure 26: Usage Model II: Single process for applications and drivers
4.2 Application Process

The goal of the application process is to allow application developers to design and debug their code with abstraction of underlying system software such as OS, drivers and firmware. It currently supports basic multi-threading, POSIX API, File I/O, System Calls, TCP/IP protocol stack and our ISS-less VP framework described in Section 3. The software stack is also easily portable to different architecture and platform by trapping the architecture and platform dependencies in a small library. To provide the basic system software features described above, our Application process (Figure 27) consists of:

- POSIX compliant standard library for POSIX API support
- Threading library for multi-threading and blocking system call implementation
- Virtual FS for File System Abstraction
- System Call API for abstracting system calls which triggers system software such as drivers and TCP/IP protocol stack
- Arch API which is the only library needs modification when porting to different architectures

In the following subsections, the detailed implementation of each layer is discussed.
<table>
<thead>
<tr>
<th>Application</th>
<th>POSIX compliant standard library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual FS</td>
<td>Threading</td>
</tr>
<tr>
<td>Arch</td>
<td>System Call API</td>
</tr>
</tbody>
</table>

**Figure 27: Application Process**

### 4.2.1 POSIX compliant standard library

This layer implements the basic standard libraries used by applications such as `printf()`, `malloc()` and `open()`. We have chosen a compact implementation of standard C library, newlib [25], which is very widely used in embedded software as our base library. Figure 28 shows a simplified architecture of newlib which consists of standard C and standard math library implementation on top of OS supporting layer called libgloss. Since one of our objectives is to create a highly portable software stack, we have abstracted away architecture dependent code such as `setjmp` and system calls by replacing with stub functions to underlying Arch API and System Call API respectively. Now our modified newlib not only support POSIX compliant standard library functions but also is completely architecture/platform independent.
Figure 28: Original newlib (left) vs modified arch/platform independent newlib (right)

4.2.2 Threading Library

To support multi-threading and blocking system calls, we have chosen a user-level non-preemptive multi-threading library called GNU Pth [26]. The main reason for the choice was in its highly portable implementation, which is a key to our software stack and its pthread API emulation layer to support pthread applications. The interaction with POSIX compliant standard library and underlying layers are shown in Figure 29.
The library consists of a scheduler to coordinate multiple threads by context switching to another thread upon self-yielding by the current thread. Any calls from system call stubs in POSIX compatibility standard library (i.e. file access and network socket access) are handled by first checking the availability of the device. If the device (file or network socket) is available, it immediately calls underlying System Call API. Otherwise, the library puts the current thread into sleep and context switch to allow another thread to run while waiting for the device to be available. This avoids a thread from blocking the entire processor when waiting for a device.

To achieve architecture independence, we have made a few modifications to the original Pth implementation including abstracting away jump buffer handling used for context switching.
4.2.3 Virtual File System

For many applications, capability of reading and writing from/to a file is an essential requirement. However implementing storage devices such as Flash in target SoC, setting up device driver for accessing the storage and even copying files to these devices require significant system engineering work. To overcome this, our software stack implements a Virtual File System which can be built directly on host File System or Network File System using 9P protocol as shown in Figure 30. This not only makes file testing easy but also potentially improving the VP simulation performance significantly which are shown in Section 5. Virtual FS is placed just under POSIX compliant standard library so that applications can access FS through standard POSIX APIs such as `open()`, `read()`, `write()` and `close()`. Underneath Virtual FS is Threading Library (section 4.2.2) which implements blocking system call to underlying System Call API described in section 4.2.2.

![Figure 30: Virtual File System Implementation on Host FS (left) and Network FS using 9P (right)](image-url)
4.2.4 System Call API

System Call API is the abstraction API for system calls to the Emulated Kernel process. It essentially hides the underlying system call details by encoding system call number and argument data into a message and sends it to Emulated Kernel Process. Emulated Kernel Process then executes the actual system call implementation on behalf of the Application. An example of System Call API message sent for write() system call is illustrated in Figure 31. The message contains unique system call ID representing write() followed by the input arguments which are variable depending on the argument type and size.

```
write(int Fd, void* Data, int Size)
```

Figure 31: System Call API message with write() system call
4.2.5 Arch

Arch contains processor architecture dependent codes found in Application process. This consists of part of POSIX compliant standard library (section 4.2.1) and threading library (section 4.2.2). This is the only library that needs to be ported when building the Application Process on a different architecture. Table 7 shows a list of functions or header files in the directory. To port Arch to a new processor, a user essentially needs to provide compatible implementations for all the functions and header files. Most of the implementations can be found from standard C library (i.e. glibc and newlib) ported for the processor. Please note that this arch serves as one variation of Arch API in section 3.3.3 and it is designed solely for the Application Process in our software stack.

<table>
<thead>
<tr>
<th>Library Needed by</th>
<th>Function or header file</th>
<th>Brief Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX Compliant Standard Library</td>
<td>Setjmp(jmpBuf), longjmp(jmpBuf)</td>
<td>Jump buffer operation for context switching</td>
</tr>
<tr>
<td></td>
<td>Endian.h</td>
<td>Defines endianness of the processor</td>
</tr>
<tr>
<td></td>
<td>App.lds</td>
<td>Linker script for building application</td>
</tr>
<tr>
<td></td>
<td>crt0.S and its associated files</td>
<td>Application start-up code</td>
</tr>
<tr>
<td>Threading Library</td>
<td>getSp(jmpBuf)</td>
<td>Return Stack Pointer location in jump buffer</td>
</tr>
<tr>
<td></td>
<td>getPc(jmpBuf)</td>
<td>Return PC location in jump buffer</td>
</tr>
</tbody>
</table>

Table 7: Arch directory contents
4.2.6 User Signals

Some applications may register a signal handler for a specific signal such as SIGIO or SIGINT to run their handler asynchronously upon the signal delivery in user application. In POSIX, functions used for registering such handler are:

- signal(int sig, void (*func)(int))
- sigaction(int sig, const struct sigaction *act, struct sigaction *oct)

Our software stack supports it by leveraging signal mechanism in host via Host Signal System Call as shown in Figure 32.

(1) At System Call API layer, signal system call encodes Process ID of the Application Process with regular arguments and sends the message to the Emulated Kernel Process. It also calls `sigaction()` on host to register the signal handler with host.

(2) Later, when the signal event has occurred, Emulated Kernel Process raises the registered signal to Application Process using the Process ID sent during signal system call, resulting in signal handler being executed asynchronously in Application Process.
Figure 32: User Signal mechanism in between Application and Emulated Kernel Processes.

4.3 Emulated Kernel Process

Contrasting to the Application Process, Emulated Kernel Process is designed for device driver developers to test their code in user-level with abstraction from upper level applications. The highest level visible to drivers are system calls from applications, hence the process contains purely low-level system software serving as system call implementations. The process (Figure 33) contains:

- System Call Poller Thread for polling system call messages arriving from Application Process using System Call API
- I/O Memory API compatible Driver API for implementing drivers
- TCP/IP Stack for serving Socket system call using Ethernet driver
- Interrupt API compatible Global ISR for external interrupt handling
4.3.1 System Call Poller

System Call Poller is a thread infinitely polling for any incoming System Call request arriving from Application Process. Upon arrival, it uses System Call API to decode the message and calls appropriate system call implementation. Most of the system calls require calling either TCP/IP stack through socket or driver functions through our Driver API (explained in section 4.4.3). Table 8 shows a few major system call implementation and their dependencies. Figure 34 illustrates how a “gettimeofday” system call is served.

(1) Application Process sends an encoded message for gettimeofday system call. System Call Poller uses System Call API to extract the arguments and call timer driver to fetch the current time.

(2) Timer driver reads a register on timer peripheral which holds the current time using I/O Memory API. I/O Memory API implicitly sends an encoded message to VP and receives a response containing the register value.

(3) The register value is converted to a gettimeofday compatible return value. The value is then encoded through System Call API and sent back to the Application Process.
### Table 8: Major system call dependencies

<table>
<thead>
<tr>
<th>System Call</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>open/close/read/write/ioctl</td>
<td>All Drivers (no file access as it was implemented in Application Process)</td>
</tr>
<tr>
<td>recv/send/connect/accept/socket</td>
<td>Network socket</td>
</tr>
<tr>
<td>select</td>
<td>Drivers and TCP/IP socket</td>
</tr>
<tr>
<td>gettimeofday</td>
<td>Timer Driver</td>
</tr>
<tr>
<td>sleep/usleep/nanosleep</td>
<td>Threading Library and Timer Driver</td>
</tr>
</tbody>
</table>

Figure 34: Example of how “gettimeofday” system call is served in Emulated Kernel Process
4.3.2 Driver API

Driver API provides set of interface functions for drivers to export to System Call Poller. The set of functions are very similar to what Linux Driver defines and are shown in Figure 35. At minimal, there are only 3 main functions that a driver needs to implement, (*read), (*write) and (*ioctl). (*ioctl) function allows variable length argument to allow custom function creation. In this driver API, it is completely up to the driver designer to implement various functions such as initialization and interrupt handler registration using (*ioctl) function. We also define a driver API for registering drivers with the Emulated Kernel Process (shown as driver_fns[]). When adding a new driver, all there is to be done is to add an entry in driver_fns[] with device name (e.g. /dev/ttyS0) followed by (*read), (*write) and (*ioctl) functions. It will be automatically visible by System Call Poller and accessed using the registered device name.

```c
struct driver_fn {
    char* name;
    int (*read)(void* buf, size_t size, void* opaque);
    int (*write)(void* buf, size_t size, void* opaque);
    int (*ioctl)(unsigned long int request, ...);
};

struct driver_fn driver_fns[5] = {
    {"/dev/ttyS0", usrp2_uart_read, usrp2_uart_write, NULL},
    {"/dev/rtc0", usrp2_timer_read, NULL, NULL},
    {"/dev/lcd0", NULL, s3c24x0_lcd_write, s3c24x0_lcd_ioctl},
    {"/dev/net0", xemaclite_read, xemaclite_write, xemaclite_ioctl},
    {"/dev/intc0", NULL, NULL, usrp2_intc_ioctl},
};
```

Figure 35: Driver API
4.3.3 TCP/IP Stack

TCP/IP network stack is required by many embedded software applications via socket system calls. Our software stack implements network stack using a light weight but fully compatible TCP/IP stack called lwIP [27]. LwIP is very widely accepted in embedded software due to its small memory footprint and socket API compatibility layer. We have ported lwIP to our software stack by implementing a platform-independent network interface on our driver API. Finally, lwIP socket API is set up to be called directly by our System Call Poller.

4.4 Example with H.264 Video Decoding using FFmpeg

We now demonstrate a real-life application and set of drivers running on our software stack. Figure 36 shows pseudo code of the application. It basically reads a H.264 movie file from File System, using FFmpeg [24] codec library to decode each frame to pixel format and display it on LCD screen via SDL [23] multi-media library. For this demo, we have set up the virtual FS to use Network File System so it also exercises TCP/IP stack and Ethernet driver. We have used exactly same drivers and peripheral models found in the custom ARM9 based SoC described in section 3.4.
4.4.1 Building Application Process

First, application, FFMpeg video codec library and SDL multimedia library, are simply compiled using host compiler. They are then linked with our Application Process pre-built libraries including x86 version of Arch. Essentially, all application developer needs to do is to compile their code as if a desktop application and link with pre-built libraries provided by our software stack.

4.4.2 Building Emulated Kernel Process

Only task required for building Emulated Kernel Process was to register drivers with our Driver API by filling in the driver table (Figure 35). When using an existing drivers, the driver functions may needed to be wrapped by our driver API function format. All memory-mapped I/O accesses have already been replaced with I/O Memory API in section 3.4.1. The drivers were then compiled using host compiler linking with other libraries in Emulated Kernel Process.
4.4.3 Starting a Network File System Server

Since we want to use Network FS to fetch a H.264 file sitting on network, we started a server hosting the encoded H.264 file called “movie.264” in another PC using 9P protocol [28]. 9P protocol server basically makes a directory in a server PC to be visible by our VP with an illusion of the FS sitting in the target platform. Files on the server can be seamlessly updated by VP issuing using Virtual FS in our software stack or directly touching files in the server directory.

4.4.4 Running the Application and Drivers on the Software Stack

All four processes, Application, Emulated Kernel, Broker and ISS-less VP, were executed on host and the application was able to successfully fetch the H.264 file from network File System, to decode the file and to display the rendered pixel frames on LCD peripheral model in VP. A life-cycle of a H.264 frame is shown in Figure 37.

1. Application sends read system call to read from file “movie.264”
2. Virtual FS implemented as 9P Network FS client processes the request and sends a socket system call to receive the file data from 9P server. The request gets encoded and sent to Emulated Kernel Process via Broker
3. System Call Poller in Emulated Kernel Process receives and decodes the socket system call message and requests TCP/IP stack to fetch data from server
4. Ethernet driver is then called to fetch an Ethernet packet from Ethernet peripheral model in QEMU. It does so by using I/O Memory API to send request to Broker
(5) Ethernet peripheral model in QEMU receives the request and fetches the actual file data from 9P FS Server via host Ethernet device.

(6) The data is then read by Ethernet driver via I/O Memory API.

(7) The data is returned to System Call Poller where it converts the data to a format compatible with the original socket system call return format.

(8) The data is encoded and sent back to Application via System Call API.

(9) Application receives file data for the input H.264 file, decodes a frame using FFmpeg and calls write system call to write the decoded pixel data to frame buffer.

(10) System Call Poller receives and decodes the write system call request and calls LCD driver.

(11) LCD driver sends the frame buffer data to Frame Buffer model in QEMU via I/O Memory API.

(12) Finally Frame Buffer model in QEMU is updated with new decoded image. At next screen refresh, the decoded pixel image was displayed successfully on host LCD screen.
Figure 37: Life time of an encoded video frame in Network FS until it is decoded and displayed on LCD peripheral model in ISS-less VP

4.5 Summary

This section discussed our software stack which allows application and device driver developers to independently develop their codes without waiting for system software delivery. The software provides framework to build application and device drivers into two separate processes, Application Process and Emulated Kernel Process for clean development environment. The chapter ended with a complete example of H.264 video decoding application.
Chapter 5
Evaluation and Analysis

5 Evaluation and Analysis

This section discusses evaluation and analysis of our software stack on the ISS-less VP framework in terms of performance, portability and user experiences. The biggest benefit of this work is to improve embedded software productivity by enabling desktop development and system software independent environment. Hence, the main objective for performance analysis is to show that ISS-less VP execution time are reasonably comparable with conventional VP.

5.1 Experiment Setup

5.1.1 Reference Virtual Prototype Setup

As described throughout this work, we have used QEMU with ISS emulating our custom SoC platform described in section 3.4 as our performance reference. QEMU was chosen as our base VP because of its open source nature and widely accepted state-of-the-art performance ISS which achieves hundreds of MIPS performance. For a fair comparison and portability analysis purpose (section 5.3), we have built two variations of the SoC platform with exactly same peripheral sets but different ISS, namely ARM9 (Figure 38 a) and Xilinx Microblaze (Figure 38 b). The two platforms were used as reference platforms representing conventional VP with ISS and compared against our ISS-less QEMU shown in Figure 38 c.
For a reference target setup, the applications were cross-compiled into both ARM and Microblaze binary and linked with Usage Model II of our software stack (Figure 26). Usage Model I (Figure 25) is not supported in conventional VP due to absence of the Broker.

For ISS-less VP, we have compiled the application with host compiler and linked with both Usage Model of our software. For usage model II, Application and Emulated Kernel processes were merged into a single process by modifying System Call API to directly call functions in System Call Poller Process (Figure 39). This potentially achieves higher performance due to elimination of System Call API message exchange between Application and Emulated Kernel Process with the sacrifice of separated development environment for application and driver developers.
Figure 39: Combined software stack with System Call API directly calling System Call Poller functions

5.1.2 Evaluation Methodology

We have run our software stack in several configurations with different applications using same driver sets and VP peripheral models. We compared runtime of ISS-less VP against the two reference SoC platforms on conventional VP (QEMU) with ARM ISS and Microblaze ISS. All experiments were run on a host with specifications shown in Table 9.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Core2 Duo @ 2.20 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>4.00 GB</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu Linux 10.04 32-bit</td>
</tr>
</tbody>
</table>

Table 9: Host Specification
The different configurations of software stack are summarized and briefly explained in Table 10.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>ISS</th>
<th>Swstack binary</th>
<th># of software stack processes</th>
<th>Virtual FS</th>
<th>Broker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target-arm</td>
<td>ARM</td>
<td>ARM</td>
<td>1</td>
<td>9P Network FS</td>
<td>None</td>
</tr>
<tr>
<td>Target-mb</td>
<td>Microblaze</td>
<td>Microblaze</td>
<td>1</td>
<td>9P Network FS</td>
<td>None</td>
</tr>
<tr>
<td>Host-1-n</td>
<td>None</td>
<td>x86</td>
<td>1</td>
<td>9P Network FS</td>
<td>IPC</td>
</tr>
<tr>
<td>Host-1-h</td>
<td>None</td>
<td>x86</td>
<td>1</td>
<td>Host FS</td>
<td>IPC</td>
</tr>
<tr>
<td>Host-2-n</td>
<td>None</td>
<td>x86</td>
<td>2</td>
<td>9P Network FS</td>
<td>IPC</td>
</tr>
<tr>
<td>Host-2-h</td>
<td>None</td>
<td>x86</td>
<td>2</td>
<td>Host FS</td>
<td>IPC</td>
</tr>
</tbody>
</table>

Table 10: Different configurations of software stack

*Target-arm / Target-mb:*

This is our performance reference platform which runs cross-compiled Usage Model II on conventional VP with ISS (Figure 40). All memory-mapped I/O and interrupts are handled by ISS within a single process. File access is implemented by 9P Network FS client to fetch a file system sitting on a remote computer in network.
Usage Model II (Figure 26) compiled and executed natively on our x86 host with our ISS-less VP. Memory-mapped I/O accesses and Interrupts are handled by I/O Memory API and Interrupt API respectively. Virtual FS are built on either 9P FS client or host FS. Broker communication is implemented using Inter Process Communication (IPC) on host.

Usage Model I (Figure 25) compiled and executed natively on our x86 host with our ISS-less VP. System Calls, Memory-mapped I/O accesses and Interrupts are handled by System Call API,
I/O Memory API and Interrupt API respectively. Virtual FS are built either 9P FS client or directly on FS on simulation host. Broker communication is implemented using IPC on host.

5.1.3 Benchmarks

We have chosen a variety of real-life applications as well as several standard benchmarks from SPEC 2000 for our applications. Each application exercises different characteristics of software stack and SoC platform which are summarized in Table 11. The variation covers applications ranging from computational intensive (i.e. Threading and GZIP) to I/O intensive (i.e. Bitmap Blitting and File Access). More complex applications with hybrid of computation and I/O are also examined (i.e. video decoding).
<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>I/O Access</th>
<th>Computation</th>
<th>Input File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threading</td>
<td>Nested threading</td>
<td>None</td>
<td>Very high</td>
<td>None</td>
</tr>
<tr>
<td>Bitmap blitting</td>
<td>Blit bitmaps on frame buffer</td>
<td>Very high</td>
<td>Low</td>
<td>None</td>
</tr>
<tr>
<td>File Access</td>
<td>Load a file</td>
<td>Very high</td>
<td>Low</td>
<td>0.3 – 3.0 MB</td>
</tr>
<tr>
<td>H.264 video decoding</td>
<td>Load a H.264 file, decode using FFMpeg and write to frame buffer</td>
<td>Very high</td>
<td>Very high</td>
<td>0.1 MB</td>
</tr>
<tr>
<td>WMV video decoding</td>
<td>Load a WMV file, decode using FFMpeg and write to frame buffer</td>
<td>Very high</td>
<td>High</td>
<td>1.0 MB</td>
</tr>
<tr>
<td>SPEC 2000</td>
<td>164.gzip</td>
<td>None</td>
<td>Very high</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>164.gzip w/ file loading</td>
<td>High</td>
<td>Very high</td>
<td>3.1 MB</td>
</tr>
<tr>
<td></td>
<td>176..gcc</td>
<td>Low</td>
<td>Very high</td>
<td>0.3 MB</td>
</tr>
<tr>
<td></td>
<td>181.mcf</td>
<td>None</td>
<td>Very high</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>181.mcf w/ file loading</td>
<td>Moderate</td>
<td>Very high</td>
<td>1.2 MB</td>
</tr>
<tr>
<td></td>
<td>255.vortex</td>
<td>High</td>
<td>Very high</td>
<td>3.4 MB</td>
</tr>
<tr>
<td></td>
<td>300.twolf</td>
<td>Low</td>
<td>Very high</td>
<td>0.5 MB</td>
</tr>
</tbody>
</table>

Table 11: Benchmark description and statistics
5.2 Performance

In this section, we show our performance results by first showing extreme case analysis in terms of processor computation, IPC and file access to provide insight into the characteristics of our ISS-less VP framework. Next, we show the complete data for every benchmark and analyze the results.

5.2.1 Processor Computation

To test how pure computational applications scale with our framework, we have run 164.gzip from SPEC2000 built on our software stack with configurations described in section 5.1.2. We ran gzip with different compression and decompression levels to vary the amount of computation executed. Only computational time on compression and decompression was collected by start measuring time after the input file was completely loaded. The results are shown in Table 12. Every level of compression/decompression in gzip resulted in at least a magnitude improvement in performance compared to both Target-mb and Target-arm. This is expected as GZIP compression and decompression results in zero I/O Memory or System Call. Hence both Host-1 and Host-2 run the GZIP algorithm within its single process software stack and Application Process respectively with no traffic through the Broker. This test essentially achieves comparison between running non-I/O software on a pure ISS and natively on host. Also note that different implementation of ISS for different processors (implemented by open source community for QEMU in this case) can potentially have significant performance difference. Our results also show Microblaze ISS runs almost 50% slower than ARM ISS. Please note that due to no file
access generated by GZIP, Host-1-h and Host-2-h configurations are not shown in the results. They achieved exactly same execution time as Host-1-n and Host-2-n respectively.

<table>
<thead>
<tr>
<th>Lv</th>
<th>Target-mb</th>
<th>Target-arm</th>
<th>Host-1-n</th>
<th>Host-2-n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.60</td>
<td>3.80</td>
<td>0.36</td>
<td>0.38</td>
</tr>
<tr>
<td>2</td>
<td>6.60</td>
<td>3.82</td>
<td>0.38</td>
<td>0.38</td>
</tr>
<tr>
<td>3</td>
<td>6.76</td>
<td>4.82</td>
<td>0.39</td>
<td>0.39</td>
</tr>
<tr>
<td>4</td>
<td>6.94</td>
<td>4.17</td>
<td>0.40</td>
<td>0.40</td>
</tr>
<tr>
<td>5</td>
<td>7.24</td>
<td>4.44</td>
<td>0.43</td>
<td>0.43</td>
</tr>
<tr>
<td>6</td>
<td>7.70</td>
<td>4.85</td>
<td>0.47</td>
<td>0.48</td>
</tr>
<tr>
<td>7</td>
<td>8.11</td>
<td>5.11</td>
<td>0.52</td>
<td>0.50</td>
</tr>
<tr>
<td>8</td>
<td>9.22</td>
<td>6.80</td>
<td>0.62</td>
<td>0.60</td>
</tr>
<tr>
<td>9</td>
<td>11.48</td>
<td>7.91</td>
<td>0.79</td>
<td>0.81</td>
</tr>
</tbody>
</table>
Table 12: GZIP performance
5.2.2 IPC

IPC through the Broker includes messages sent and received using

- I/O Memory API
- System Call API

This communication latency is the major performance trade-off in our framework. We have demonstrated the effects of the latency using a Bitmap Blitting application. Bitmap Blitting is an operation to blend a pixel image into the frame buffer to be displayed on LCD display. The operation requires heavy memory-mapped I/O to write blended pixel data to the frame buffer sitting in VP. In Target-arm and Target-mb, this is simply a write to a section of system memory configured for frame buffer. In Host-1, it requires I/O Memory API to send the pixel datas to VP through Broker. In Host-2, on top of I/O Memory API, System Call API needs to first deliver the pixel data from Application Process to Emulated Kernel Process shown in step (9) – (11) in Figure 37, essentially doubling the inter process communication traffic.

The results are shown in Table 13. While Host-1 suffers up to 57% compared to Target-arm performance, Host-2 suffers further up to 13%. These results show the worst slow-down possible for our framework.
<table>
<thead>
<tr>
<th>Size(MB)</th>
<th>Target_mb</th>
<th>Target-arm</th>
<th>Host-1</th>
<th>Host-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.42</td>
<td>0.13</td>
<td>0.08</td>
<td>0.16</td>
<td>0.58</td>
</tr>
<tr>
<td>14.83</td>
<td>0.26</td>
<td>0.16</td>
<td>0.30</td>
<td>1.15</td>
</tr>
<tr>
<td>22.25</td>
<td>0.39</td>
<td>0.23</td>
<td>0.40</td>
<td>1.71</td>
</tr>
<tr>
<td>29.66</td>
<td>0.51</td>
<td>0.31</td>
<td>0.54</td>
<td>2.38</td>
</tr>
<tr>
<td>37.08</td>
<td>0.64</td>
<td>0.39</td>
<td>0.67</td>
<td>2.84</td>
</tr>
<tr>
<td>44.49</td>
<td>0.77</td>
<td>0.46</td>
<td>0.81</td>
<td>3.47</td>
</tr>
<tr>
<td>51.91</td>
<td>0.90</td>
<td>0.54</td>
<td>0.93</td>
<td>4.16</td>
</tr>
<tr>
<td>59.33</td>
<td>1.03</td>
<td>0.62</td>
<td>1.05</td>
<td>4.54</td>
</tr>
<tr>
<td>66.74</td>
<td>1.16</td>
<td>0.69</td>
<td>1.19</td>
<td>4.92</td>
</tr>
<tr>
<td>74.16</td>
<td>1.28</td>
<td>0.77</td>
<td>1.34</td>
<td>5.70</td>
</tr>
</tbody>
</table>
Table 13: Frame Buffer Write Performance
5.2.3 File Access Performance

Another key performance factor in our software stack is accessing Virtual File System for reading and writing a file. For Target-arm and Target-mb only 9P Network FS is available while Host-* configurations supports host FS as well. We have demonstrated File Access performance by an application that simply open and read variable size from a file on a Virtual FS. The performance is shown in Table 14. When using Network FS, Host-1-n and Host-2-n have very similar performance and are about 20% slower than Target-mb and Target-arm. This is due to System Call API and I/O Memory API to send socket system call to QEMU Ethernet peripheral model. As well, the fact that Host-1 and Host-2 result in almost identical performance suggests that TCP/IP access to Network FS server by far dominates the latency of IPC. When Host FS is used in Host-1-h and Host-2-h, the performance achieves very close to native file access on host as it completely eliminates System Call API and I/O Memory API calls to fetch files from network FS server. The performance improvement achieved is in three orders of magnitude.
<table>
<thead>
<tr>
<th>Size(MB)</th>
<th>Target-mb</th>
<th>Target-arm</th>
<th>Host-1-n</th>
<th>Host-2-n</th>
<th>Host-1-h</th>
<th>Host-2-h</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>8.90</td>
<td>9.00</td>
<td>11.18</td>
<td>11.27</td>
<td>0.31</td>
<td>0.35</td>
</tr>
<tr>
<td>0.6</td>
<td>17.71</td>
<td>17.71</td>
<td>21.38</td>
<td>21.69</td>
<td>0.93</td>
<td>0.16</td>
</tr>
<tr>
<td>0.9</td>
<td>26.52</td>
<td>26.55</td>
<td>32.96</td>
<td>33.01</td>
<td>0.12</td>
<td>0.22</td>
</tr>
<tr>
<td>1.1</td>
<td>35.33</td>
<td>35.91</td>
<td>43.55</td>
<td>43.58</td>
<td>0.20</td>
<td>0.32</td>
</tr>
<tr>
<td>1.4</td>
<td>44.13</td>
<td>44.05</td>
<td>54.98</td>
<td>54.84</td>
<td>0.25</td>
<td>0.37</td>
</tr>
<tr>
<td>1.7</td>
<td>52.96</td>
<td>52.61</td>
<td>66.25</td>
<td>66.99</td>
<td>0.93</td>
<td>0.62</td>
</tr>
<tr>
<td>2.0</td>
<td>61.76</td>
<td>62.50</td>
<td>76.89</td>
<td>76.86</td>
<td>0.34</td>
<td>0.96</td>
</tr>
<tr>
<td>2.3</td>
<td>70.58</td>
<td>70.52</td>
<td>88.00</td>
<td>88.09</td>
<td>0.15</td>
<td>0.70</td>
</tr>
<tr>
<td>2.6</td>
<td>79.45</td>
<td>80.00</td>
<td>99.68</td>
<td>100.05</td>
<td>0.42</td>
<td>0.71</td>
</tr>
<tr>
<td>2.9</td>
<td>87.94</td>
<td>87.89</td>
<td>110.20</td>
<td>110.20</td>
<td>0.62</td>
<td>0.85</td>
</tr>
</tbody>
</table>
Table 14: File Access Performance
5.2.4 Benchmark Performance

With three performance characteristics analyzed, we ran our real-life benchmarks and SPEC benchmarks. Real-life benchmark results are shown in Table 15. Nested multithreading (A in Table 15) results in significant performance improvement as threading was executed internally in the process where application sits resulting in no System Call API or I/O Memory API access. In contrast, majority of Bitmap Blitting (B in Table 15) codes are writing blitted pixel image to Frame Buffer sitting in VP as described in section 5.2.2, performance was degraded going from Target, Host-1 to Host-2. H.264 and WMV decoding were interesting applications as they contain hybrid of heavy computation (video decoding), intensive I/O (frame buffer write) and massive file access (loading encoded video file). We have configured the application to two variations with one fetching and decoding the input movie file (C and E in Table 15) while the other also writes the decoded pixels to the frame buffer sitting on VP (D and F in Table 15). Results show that with no frame buffer operation, both H.264 and WMV with ISS-less VP outperforms both Target-mb and Target-arm. With Frame Buffer writing included, much more I/O traffic is introduced. Host-1 slows down slightly by about 10 % while Host-2 suffers significantly by about 200-300% due to increased IPC between processes. However, most configurations are still competitive with the reference performance (i.e. F in Table 15 is only about 2 times slower than Target-arm and 1.5 times slower than Target-mb). Finally with Virtual FS using Host FS (Host-1-h and Host-2-h), performance improves significantly up to 15 times (observed in E in Table 15 compared against Target-mb). This also proves the File Access performance dominance in Software Stack.
## Real-life benchmark performance (time in sec)

<table>
<thead>
<tr>
<th></th>
<th>Target-mb</th>
<th>Target-arm</th>
<th>Host-1-n</th>
<th>Host-1-h</th>
<th>Host-2-n</th>
<th>Host-2-h</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Nested</td>
<td>1.86</td>
<td>1.16</td>
<td>0.31</td>
<td>0.31</td>
<td>0.34</td>
<td>0.32</td>
</tr>
<tr>
<td>Multithreading</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B. Bitmap</td>
<td>1.34</td>
<td>0.90</td>
<td>1.50</td>
<td>1.50</td>
<td>6.82</td>
<td>6.81</td>
</tr>
<tr>
<td>Blitting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C. H.264 Video</td>
<td>7.10</td>
<td>4.21</td>
<td>2.89</td>
<td>0.81</td>
<td>2.76</td>
<td>0.88</td>
</tr>
<tr>
<td>Decoding</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D. H.264 Video</td>
<td>9.92</td>
<td>5.02</td>
<td>3.11</td>
<td>1.84</td>
<td>9.00</td>
<td>7.85</td>
</tr>
<tr>
<td>Decoding +</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FrameBuffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E. WMV Video</td>
<td>13.10</td>
<td>9.14</td>
<td>10.21</td>
<td>0.80</td>
<td>9.98</td>
<td>0.82</td>
</tr>
<tr>
<td>Decoding</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F. WMV Video</td>
<td>16.02</td>
<td>9.98</td>
<td>11.99</td>
<td>3.12</td>
<td>22.15</td>
<td>10.94</td>
</tr>
<tr>
<td>Decoding w/ 9P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FS +</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FrameBuffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 15: Real-life application performance

SPEC2000 benchmark performance is shown in Table 16. With most of benchmarks consisting heavy computational kernel, input file loading and output file writing, our Host-1-h and Host-2-h achieve significant speedup up to 90 times observed in 300.twolf. Network FS implementation (Host-1-n and Host-2-n) also achieve observable improvement because of the heavy computation hiding the small slow-down in File Access discussed in section 5.2.3. For 164.gzip and 181.mcf, we were able to measure only the time spent on its computational kernel without input file loading and output file writing and observed about 9-20 time speedup as well.
<table>
<thead>
<tr>
<th>SPEC 2000 benchmark performance (time in sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>164.gzip</td>
</tr>
<tr>
<td>164.gzip w/ file loading</td>
</tr>
<tr>
<td>176.gcc</td>
</tr>
<tr>
<td>181.mcf</td>
</tr>
<tr>
<td>181.mcf w/ file loading</td>
</tr>
<tr>
<td>255.vortex</td>
</tr>
<tr>
<td>300.twolf</td>
</tr>
<tr>
<td>300.twolf w/ file loading</td>
</tr>
</tbody>
</table>
Table 16: SPEC2000 benchmark performance

5.3 Software Stack Portability

In this section, we discuss our experience in porting the software stack to different architectures, ARM and Microblaze. This is an important metric as our objective is to create a highly portable software stack for application and device driver developers to setup with minimal project overhead. Porting to both architectures were done by solely providing implementations for functions and header files in Arch library and compiling the entire software using the target processor cross-compiler. Table 17 shows the actions taken for implementing the Arch library for ARM and Microblaze. The porting took only one day while testing took two days.
<table>
<thead>
<tr>
<th>Function or header file</th>
<th>ARM</th>
<th>Microblaze</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setjmp(jmpBuf), longjmp(jmpBuf)</td>
<td>Used setjmp and longjmp assembly implementation found in ARM port of newlib</td>
<td>Used setjmp and longjmp assembly implementation found in Microblaze port of newlib</td>
</tr>
<tr>
<td>Endian.h</td>
<td>Defined BYTE_ORDER as LITTLE_ENDIAN</td>
<td>Defined BYTE_ORDER as BIG_ENDIAN</td>
</tr>
<tr>
<td>App.lds</td>
<td>Based on default linker script of ARM cross-compiler and adjusted heap size to suite application</td>
<td>Based on default linker script of Microblaze cross-compiler and adjusted heap size to suite application</td>
</tr>
<tr>
<td>Crt0.S and its associated files</td>
<td>Based on crt0.S in ARM port of newlib and added Interrupt Vector Table setup and kernel stack initialization</td>
<td>Based on crt0.S in Microblaze port of newlib and modified stack pointer address to suite our custom SoC platform</td>
</tr>
<tr>
<td>getSp(jmpBuf)</td>
<td>Return address (jmpBuf address + 36) where Stack Pointer is stored in ARM jump buffer</td>
<td>Return address (jmpBuf + 0) where Stack Pointer is stored in Microblaze jump buffer</td>
</tr>
<tr>
<td>getPc(jmpBuf)</td>
<td>Return address (jmpBuf address + 40) where Program Counter is stored in ARM jump buffer</td>
<td>Return address (jmpBuf + 12) where Program Counter is stored in Microblaze jump buffer</td>
</tr>
</tbody>
</table>

Table 17: Tasks for implementing the Arch library for ARM and Microblaze
5.4 User Experience

Key objective of this work is to improve embedded software development productivity by enabling rich desktop development tools. This section discusses number of development experience improvement using the framework.

Being able to use memory-analysis tools such as Valgrind was one of the strong benefits the framework provides. Using these tools, we were able to find number of memory leaks in applications we were testing on our framework. As a concrete example, desktop tools helped to debug a tough problem with uninitialized field in a C struct. Figure 41 shows an incorrect initialization of sigaction structure when registering a signal handler with a signal. When running this code on ISS, mysterious behavior occurs upon a signal delivery where Stack Pointer and Program Counter point to an invalid memory address resulting in execution corruption. This kind of problem is non-trivial to debug especially when developing embedded software. However, using desktop memory analysis tool on our application, it immediately spotted the uninitialized C struct content and gave us a significant hint to spot the problem.
Another experience is working with GNU Debugger (GDB) that is cross-compiled for an embedded processor. Due to its low maturity of the GDB ported for the architecture, some of the GDB features did not work correctly. By using our framework, we were able to use mature full-blown x86 GDB on the host which none of the portability issues were observed.

Figure 41: uninitialized C struct example with sigaction system call
Chapter 6
Conclusion and Future Work

6 Conclusion and Future Work

From what has been discussed and demonstrated, we believe that ISS-less VP methodology is a promising candidate for early embedded software development. The framework brings benefits which directly improve the engineering productivity:

- Removing dependency on system software in application and driver development
- Removing system software bugs
- Saving significant overhead of developing and debugging ISS for a new processor
- Achieving desktop software development environment for embedded software
- Demonstrating that computational intensive applications on ISS-less VP can achieve performance improvement up to 2000% over conventional VP with state-of-the-art ISS performance
- Demonstrating that I/O intensive applications suffer performance loss but still achieve reasonable (within a magnitude) execution time compared to conventional VP

For future work, we plan to add a new usage model to our software stack by compiling Application, Emulated Kernel and ISS-less VP processes all into one process. This usage model can generate a single host executable to execute standalone, resulting in elimination of broker and IPC. This is predicted to achieve much higher execution performance because it completely removes slow-down factors in our framework, namely IPC communication. Although it may make software debugging more challenging by building software and hardware all into one image, the faster execution could be useful for large software regression tests at later stage of
software development. This is predicted to require work in intelligent linking between our software stack and VP process. For example, since both processes use POSIX APIs but with different implementations (i.e. POSIX compatible standard library in our software stack and host system calls made by VP), the final single process requires to link two different implementations of same POSIX library into one executable. This requires some work in GNU loader to make it happen.
References


[36] Incisive Enterprise Simulator,

[37] Innovator,


[41] A. Garcia, Comparing virtualization software performance: QEMU vs UML vs KVM,

