COISA: A Compact OpenISA virtual platform for IoT devices

Carlos Eduardo Millani[1], Alisson Linhares[1], Rafael Auler[1], Edson Borin[1]

[1]Institute of Computing – University of Campinas (UNICAMP)
Av. Albert Einstein, 1251 – 13083-852 – Campinas – SP – Brazil

{carloseduardomillani,arescarv}@gmail.com{auler,edson}@ic.unicamp.br

Abstract. *In face of the high number of different hardware platforms that we need to program in the Internet-of-Things (IoT), Virtual Machines (VMs) pose as a promising technology to allow a program once, deploy everywhere strategy. Unfortunately, many existing VMs are very heavy to work on resource-constrained IoT devices. We present COISA, a compact virtual platform that relies on OpenISA, an Instruction Set Architecture (ISA) that strives for easy emulation, to allow a single program to be deployed on many platforms, including tiny microcontrollers. Our experimental results indicate that COISA is easily portable and is capable of running unmodified guest applications in highly heterogeneous host platforms, including one with only 2 kB of RAM.*

1. Introduction

Current computer science trends suggest a future where platform heterogeneity is gracefully embraced: with Internet-of-Things (IoT) [17, 32], we will need an increased capacity of programming wildly different devices, each exploring distinct energy-performance trade-offs, yet still remain productive enough to make this industry viable from the software engineering standpoint.

There are two distinct development models engineers can explore: (1) the software is custom designed to each device, and all software development is tied to the hardware platform, or (2), software is written once for a virtual platform, but it runs on all devices in a hardware-independent fashion due to the use of virtual machine technology.

The solution (1) may not be a cost-effective choice in face of the high number of different hardware platforms that may support IoT devices (such as different versions of Arduino [16], ARM [23] and Quark [24]). For example, the Android Dalvik virtual machine powered a large slice of the mobile phone revolution because it is not attractive for programmers to learn how to program a single device if they can learn how to program a myriad of Java-running devices. Likewise, for companies, it is not advantageous to spend resources developing software locked to a single device if they can target a wider audience with virtual machine technology. If virtual machines already proved to be an effective technology in this realm, we can expect the IoT to be an important target for virtual machines because it will feature an even deeper degree of heterogeneity in comparison with the mobile industry.

However, a virtual platform has inherent overheads associated with interpretation and, in the case of Java, garbage collection, which contrasts with the scant resources typically available in IoT devices. Thus, special virtual machines must be implemented to

The authors would like to thank the support from TecSinapse, CAPES, CNPq and FAPESP grants 2011/09630-1 and 2012/50732-5.

allow its feasibility. We performed a survey of virtual machines available for constrained hardware platforms, which includes very low memory devices and microcontrollers, and out of the 10 systems we were able to find, 8 of them are focused on Java.

Instead of relying on the complex and more abstract Java ISA, virtual machines can also work with low-level (machine) code and can have an uniform binary (and bug) compatibility across all targets. The attractiveness of low-level virtual machines for resource-constrained devices comes from the higher platform simplicity of such systems: we do not need the Java Runtime Environment to support the Java ecosystem, but only a simple interpreter because the guest program is already compiled and optimized. The VM now only has to focus on how to bridge the differences between host and guest hardware platforms. Therefore, it is important that the guest ISA be as easy to be emulated as possible.

In this paper, we use MIPS-I as a guest ISA for IoT devices and present a lightweight process virtual machine that is capable of emulating MIPS binaries on several distinct platforms, including Arduino Uno [13], which features only 2 kB of RAM. Our contributions are the following:

- We show that the memory footprint of a simple interpreter can be as low as 348 bytes of RAM, and yet implement a complete, general purpose virtual machine;
- We argue that this virtual machine can be easily ported to different host systems and we successfully port it to 6 distinct platforms;
- We analyze how the simple, interpreted virtual machine varies its performance when running on hosts ranging from microcontrollers to high-end processors.

This paper is organized as follows. Section 2 presents related work in literature, Section 3 gives an overview of the COISA virtual platform, Section 4 discusses our experimental results and Section 5, our conclusions and future work.

2. Related Work

Nowadays, in consequence of the wide range of micro-controllers configurations and specifications, it is difficult to achieve a complete software portability and hardware abstraction in this realm. One way to achieve software portability is through the use of virtual machine technologies. A virtual machine is a computer program that emulates an interface to another application. This technology is present in many computer systems and has been used from the support of high-level programming languages, such as the Java virtual machine, to the implementation of processors with integrated hardware and software design, such as the Transmeta's Efficeon processor [26].

There are several virtual machines that were developed to run on resource-constrained devices [1, 8, 9, 10, 11, 19, 25]. For instance, Darjeeling [19], Simplen RTJ [9], uJ [11] and NanoVM [10] are bare metal High-Level Language (HLL) VMs that are capable of running on a limited hardware environment, such as the ones powered by 8 or 16-bit microcontrollers with 2 to 10 kB of RAM and 32 kB of flash memory. Even though these implementations focus on running Java applications, they are very simple and only capable of executing a subset of java-bytecode.

Because of the Java complexity and limited resources, some projects, like TakaTuka [12], use a compact file format to reduce the memory overhead instead of the

traditional .class file. These files contain only essential information about the runtime. TakaTuka also employs a byte-code compression and optimization that improves the load performance and execution.

One disadvantage of HLL virtual machines is that most implementations are bound to a single high-level programming language. A solution with more freedom is the IBM Mote runner [20]. The IBM Mote runner is a virtual machine designed to run on embedded systems and can be a target to all strictly-typed programming languages, such as Java and C#. This virtual machine runs a specialized bytecode, called Mote Runner intermediate language (SIL), and uses a stack-based approach to achieve a more compact implementation.

An important difference of our work against others that use a specialized byte-code for restricted platforms is that we are in favor of using a single binary format across all machines. In this sense, working with a different intermediary file specially crafted for restricted platforms defeats the purpose of using VM technology, which is to have an architecture-neutral format. In our work, we stick with a register-based approach instead of the Mote runner choice because we believe that a register-based intermediate representation is more suitable for optimizations for higher-end hosts [28].

CILIX [31] is an HLL VM for the Common Intermediate Language (CIL) that runs on resource-constrained devices, requiring 8 or 16-bit CPUs, 4 kB RAM, and 32 kB of flash memory. Thanks to the CIL, the CILIX is compatible with many programming languages, such as J++, C#, Visual Basic, C++ and F#.

Table 1 presents a summary of virtual machines that are compatible with devices with low RAM availability.

Project	CPU (bits)	RAM (kB)	Flash (kB)	ISA	Language
SimpleRTJ [9]	8/16/32	2-24	32-128	Java Bytecode	Java
Darjeeling [19]	8/16	2-10	32-128	Java Bytecode	Java
NanoVM [10]	8	1	8	Java Bytecode	Java
uJ [11]	8/16/32/64	4	80-60	Java Bytecode	Java
TakaTuka [12]	8/16	4	48	Java Bytecode	Java
Squawk [29]	32	512K	4000	Squawk Bytecode	Java
IBM Mote Runner [20]	8/16/32	4K	32	SIL	C#, Java
CILIX [31]	8/16	4K	32	CIL	C#
PyMite [1]	8/16/32	5K	64	Python Bytecode	Python

Table 1. Virtual Machines comparison

HLL virtual machines typically execute an intermediate language representation that reflects important features of a specific class of languages. On one hand, this approach includes more semantic information from the source program, allowing the system to perform more aggressive runtime optimizations to improve the system performance. On the other hand, this approach also makes the virtual machine more dependent on source languages.

Different from HLL virtual machines, ISA virtual machines are capable of running low-level binary programs that were compiled for a given instruction set architecture, or ISA. This solution is typically employed to grant intrinsic compatibility with real processor implementations, allowing full binary compatibility when running the binary on systems with different ISAs.

Since processor ISAs are language agnostic, using an ISA VM instead of an HLL

VM allow us to leverage any language that has a compiler compatible with the ISA to produce portable binary code. The GCC [6], for instance, has front ends for C, C++, Objective C, Fortran, Java, Ada, and Go and can be used to produce x86, ARM or MIPS-1 compatible binaries, among others. Therefore, it is possible to create an emulator for one target of GCC and emulate it with a virtual machine.

In our research, we could find several projects that implements real ISA virtual machines [3, 4, 5, 2]. Most of them are native system machines that emulate other computer hardware, including the entire instruction set. However, we could not find a process virtual machine that implements a user-level instruction set and is able to run on a resource-constrained device such as the ATmega328 microcontroller. To this end, we propose COISA, a Constrained OpenISA process virtual machine that emulates a MIPS-1 compatible user-level instruction set and can be executed by a microcontroller with less than 2 kB of RAM. Our solution stands out from other work by providing a total independence of the language and by providing intrinsic compatibility with a processor implementation. Moreover, we show that COISA is portable by executing it on different architectures, including micro-controllers and general purpose CPUs. The next section presents the architecture of COISA.

3. The COISA Virtual Platform

COISA is a virtual platform that is composed of three main components: a Virtual Machine (VM), a Hardware Abstraction Layer (HAL) and a Thing Monitor (TM). Figure 1 illustrates the COISA virtual platform running on an ATmega328 microcontroller.

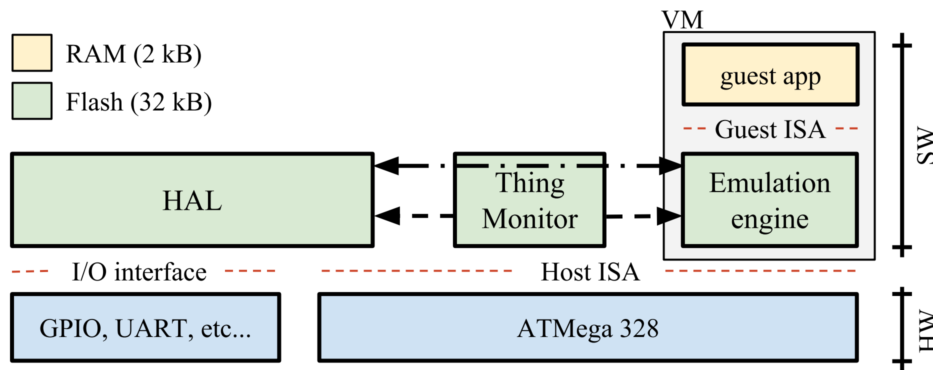

Figure 1. Overview of the COISA virtual platform

The VM is a process virtual machine [30] that is responsible for emulating guest programs, compiled for the guest ISA. It emulates user-level instructions, allowing the guest program to access guest architectural registers and the guest memory. I/O operations are performed via *syscall* instructions, which are handled by the Virtual Machine instead of a guest operating system. This organization allows us to build a very compact software stack, which in turns consumes very little memory. In fact, our experiments indicate that COISA takes only 6 kB of Flash memory and 348 bytes of RAM when executing on an ATmega328 microcontroller, leaving almost 1.7 kB out of the 2 kB RAM to the guest program.

The HAL module abstracts the underlying hardware, enabling the other modules to probe and control the hardware peripherals in a portable way. As an example, it allows

other modules to probe for existing sensors and to read data produced by them. The guest program may probe and read sensor values by performing system calls, which in turn are redirected by the VM and serviced by the HAL.

The Thing Monitor, or TM, allows remote users or devices to inspect and control the system. It is also responsible for loading the guest app into the VM memory and controlling the VM execution flow, including starting, stopping and stepping through guest instructions.

All the modules require memory resources to store their respective code, constants and variables. Since their code and constants are immutable, it is possible to store them on Flash memory, when available. This approach reduces the pressure on the main memory (typically RAM), enabling the virtual machine to dedicate most of the main memory to the guest application.

The goal with this modular infrastructure it to improve portability, allowing the virtual platform to be easily adapted to multiple environments, even to ones with scarce resources, such as a microcontrollers. Moreover, the modules of the platform are written in C and make no use of C library functions, which allows it to be compact and easily compilable to a wide variety of hardware platforms. In fact, as we discuss in Section 4.3, only fractions of the Thing Monitor and the HAL had to be changed to adapt COISA to different host platforms. For instance, the TM was the same for the three platforms with an operating system and only about ten lines of code were changed for each other one.

The VM emulation engine, which is responsible for executing the guest applications via guest ISA emulation, relies on a compact interpreter that fetches, decodes and executes guest instructions one by one. Interpreters provide lower performance than dynamic binary translators, however, they allow us to build compact and portable virtual machines [30]. Moreover, for systems that require high performance, it is possible to replace the current emulation engine by one that employs a high performing dynamic binary translator, which may be capable of achieving near native performance, as discussed by Auler and Borin [14]. In this sense, we envision an IoT with different flavors of the COISA virtual platform, ranging from very compact to high performing systems, each one requiring different amounts of hardware resources, but providing transparent software compatibility across a wide variety of hardware platforms.

3.1. Guest ISA

The guest ISA is the main interface between the virtual platform and the guest applications. This interface may affect the system in several aspects, such as:

- *guest code size*: very simple instruction set architectures may require less bits to encode instructions but may also require more instructions to code programs. Therefore, the set and the encoding of instructions may cause a big impact on the guest code size. Since we are targeting highly constrained hardware devices, such as the ATmega328 microcontroller, the guest code size must be compact.
- *emulation performance*: the emulation performance depends on the emulation technique and the guest ISA [14]. Since performance is important for certain applications, it is important to select a guest ISA that allows high performance emulation.
- *emulation complexity*: complex guest instruction set architectures may require complex and larger software engines to emulate the guest apps. In this sense, the

guest ISA must be clean, allowing the design and implementation of compact and simple emulation engines for resourced-constrained hardware platforms.

Our previous experience with ISA emulation and design [27, 14, 18] have shown us that a clean ISA, similar to MIPS-1, allows us to build a high performance emulator capable of emulating guest applications on ARM and x86 host processors at near native performance. As a result, we are designing OpenISA, an ISA that aims to be emulation friendly, empowering most of the devices in the world to execute the same set of applications. In this sense, we envision an IoT powered by platforms that execute OpenISA applications.

Since the OpenISA instructions encoding is not fully defined yet, we employ the MIPS-1 ISA as the guest ISA in our experiments and show that it is possible to build compact guest applications and a compact emulation engine for this ISA. Since MIPS-1 and OpenISA are similar in functionality and number of instructions we conjecture that OpenISA would also enable the construction of compact guest applications and a compact emulation engine, satisfying all the desired characteristics for a guest ISA.

4. Experimental Results

We evaluate the COISA virtual platform in several aspects. First, we evaluate the guest applications memory footprint. Then, we discuss the amount of memory required by the COISA virtual platform. After this, we discuss the COISA portability. Finally, we evaluate its emulation performance.

4.1. Guest code memory footprint

The guest application, including the code, is stored at the virtual machine guest memory, which may be a very scarce resource in some devices (e.g. 2 kB at the Arduino development board). Thus, the guest ISA must allow guest application to be coded in a compact way.

To evaluate the size of guest applications, we used the following subset of the programs from the Great Computer Language Shootout benchmark [15]: Ackermann, which computes the Ackermann's Function; Array, which computes the sum of two arrays; Fibonacci, which recursively computes the 12th number of the Finonacci sequence – we used a low input number to avoid overflowing the guest stack on the Arduino development board; Heapsort, which sorts a random set of integers; Lists, which performs assorted operations on linked lists; Matrix, which multiplies two 3x3 matrices – again, the input quite small to fit in RAM; Random, a random number generation function using the same seed; and Sieve, which computes the Sieve of Eratosthenes to find all primes from 1 to 768. These programs exercise a CPU-intensive workload, presenting a perfect fit to make initial tests on our general-purpose virtual machine because they stress the interpreter implementation instead of other, unrelated subsystems.

Since the benchmark applications are written in C, we utilized the GNU GCC and the GNU Binutils [7] toolchains to compile guest applications and evaluate their size. First, we compile and assemble the guest source code into object files. Then, we link these files with the crt0.o file, which contains code to initialize the stack, call the guest main function and, at the return of the guest main function, call the exit *syscall*. This file also contains a small function to allow guest programs to make system calls. After

everything is linked together, we use the `objcpy` utility to strip out all extra semantic information from the ELF file, reducing its final size. Figure 2 illustrates the flow used to compile the guest code in our experiments.

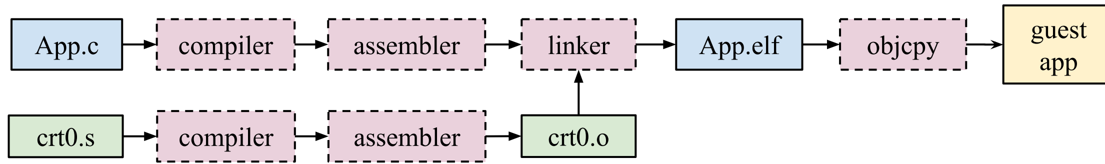

Figure 2. Guest code compilation flow

In our experiments, we did not link the guest code with the C library, since it would enlarge the guest application with functions and data that would not be required. Instead, we replaced the calls to the `printf` function by calls to two different functions: one responsible for printing strings and other responsible for outputting numbers. Those two functions invokes a *syscall* that interacts with the available hardware on the platform and prints what was ordered. This way we could check the output of the benchmarks.

For some of the applications, we also changed the benchmark input to reduce the amount of stack space required during execution. This is the case for Fibonacci and Ackermann, which can execute a deep recursion depending on the input value. Table 2 shows the size of the benchmark applications before and after stripping out the extra semantic information from the ELF file and the amount of memory required by the application stack during execution.

	Ackermann	Array	Fibonacci	Heapsort	Lists	Matrix	Random	Sieve
Original binary size	5164	5204	5132	5692	7724	6880	5344	5236
Stripped binary size	321	364	297	696	1996	1596	408	394
Minimum stack size required	696	824	376	104	80	72	24	800

Table 2. Benchmarks footprint in Bytes (`Os` optimization flag)

As we can see from Table 2, it was possible to encode all the applications with less than 2 kB. IoT systems that run on resource-constrained platforms are expected to implement simple control logic that reads sensors, provides data to other components and control peripherals. Based on the aforementioned results, we conjecture that the MIPS or an equivalent ISA is compact enough to enable coding of simple control logic with less than two kilobytes. Moreover, specialized routines designed to read or control certain peripheral can be embedded into the HAL module and made available to the guest application as a service through system calls, reducing even more the guest application size.

4.2. COISA memory footprint

We also evaluate the amount of memory resources required by COISA on six different platforms, listed in Table 3.

The AVR ATmega328 and Native x86 platforms run the COISA virtual platform without any support from host operating systems. For the AVR ATmega328 platform, we use the avr-g++ 4.8.1 cross compiler and the Arduino tool chain to compile and write the

	OSX AMD64	Linux AMD64	Linux x86
Processor Frequency	2.8 GHz	2.8 GHz	2.8 GHz
Processor Model	Intel Core i5	Intel Core 2 Duo E7400	Intel Core 2 Duo E7400
Memory	8 GB	2 GB	2 GB
	Linux ARM v7	AVR ATmega328	Native x86
Processor Frequency	1.4 GHz	16 MHz	2.8 GHz
Processor Model	ARM Cortex-A9	ATmega328-PU	Intel Core 2 Duo E7400
Memory	1 GB	2 kB RAM + 32 kB Flash	2 GB

Table 3. Platforms used in our experiments

COISA virtual platform at the ATmega328 flash memory. For the Native x86 platform, we use the Native Kit [21] to produce a bootable flash drive that automatically loads the COISA virtual platform into the main memory, sets up the x86 processor and starts the system. The other platforms contain host operating systems that enable COISA to be compiled and executed as any other ordinary host application.

The VM guest memory was adjusted based on the amount of memory available on the host platform and in the most limited environment, powered by an ATmega328, it took only 6 kB of flash memory and 348 bytes of RAM (141 bytes for data and 207 bytes for the stack), leaving 1700 out of the 2048 bytes of RAM to the guest application. Table 4 shows the memory footprint of the COISA virtual platform on all platforms tested.

	OSX AMD64	Linux AMD64	Linux x86	Linux ARM v7	AVR ATmega328	Native x86
VM Memory	5120	5120	5120	5120	1700	5120
.data Section	4	612	308	324	4	2488
.bss Section	64	276	276	136	141	388
.text Section	2939	5232	4516	6739	5784	91401
Stack	8192	8192	8192	8192	207	8192

Table 4. COISA footprint on different devices (`Os` optimization flag)

4.3. COISA portability and validation

Our prototype was carefully coded so that the amount of code that needs to be changed in order to port it to a completely new environment is small. Even through the amount of changes required to port the COISA virtual platform to new platforms may vary, in our experiments, we only had to rewrite about ten lines of code of the TM module and around 20 lines of code of the HAL module in order to port COISA to run on the AVR ATmega328 platform and the same amount to execute it on the Native x86 platform.

We used different compilers to build the COISA virtual platform in each host platform. Since our platforms rely on different underlying processor architectures, we could not use a single compiler version as a guarantee of equal code quality among them because each compiler backend may present substantial differences in maturity. However, we tried to stick with popular compilers in each case. On the OSX AMD64 platform, Xcode (LLVM-based) 6.1.0 was used, and for both Linux x86 and Linux AMD64 platforms, GCC 4.9.2 was used. GCC 4.7.3 compiled COISA to execute on the Linux ARM v7 platform, and G++ 4.8.1 on the AVR ATmega328. The Native Kit [21] used GCC 4.9.1 to build the COISA virtual platform for the Native x86 host platform.

Once compiled, the benchmark applications were loaded and executed properly by the COISA virtual platform in each one of the platforms listed in Table 3. The only exception was the application Lists on the AVR ATmega328 platform. We could not run

this application on this platform because it requires more memory than what is available at the virtual machine guest memory. All programs produced the same results in all platforms and showed that our artifact worked as planned.

The platforms used in our validation are very heterogeneous. For example, the first one contains a 64-bit 2.8 GHz Intel Core i5 running Mac OS X with 8 GB of RAM, with plenty of memory. We also have an 32-bit ARM based platform running at 1.4GHz with 1GB of RAM, being a middle-end example. The lowest end platform in our experiments is the AVR ATmega328, using a 8-bit microcontroller clocked at 16 MHz with only 2 kB of RAM.

4.4. COISA emulation performance

In order to evaluate the performance of our portable interpreter on the aforementioned host platforms, we measured the time spent to execute the guest code, the number of guest instructions emulated per second in MIPS (millions of instructions per second) and the number of host instructions per guest instructions (except for the OSX AMD64 and the Native x86 platforms).

Since the runtimes of the benchmarks are quite small in our fastest platforms, we repeated the experiments 100 times and computed the average, the standard deviation and the 95% confidence interval for each one of the experiments. Experiments with the AVR ATmega328 and the Native x86 platforms presented almost no variation while experiments with other host platforms, which contain host operating systems, presented some variation. The highest variation was verified on the experiments with the Lists application on the OSX AMD64 platform, which took, on average, 18μs to run and the 95% confidence interval was 1.37μs (7.6% of the average). We also inspected our data histograms and verified that the majority of the data is concentrated near the average. As we discuss later, this variation does not affect the conclusions derived from our analysis.

We collected results for a variety of compilation flags, but for the purposes of this analysis we focus on the tests with the `Os` compiler flag, which instructs the compiler to optimize the guest code for size. This simulates the scenario with constrained memory resources where it is important to save as much space as possible.

Table 5 shows the number of guest instructions emulated per second in MIPS (millions of instructions per second) when executing the benchmark applications on each one of the host platforms.

	OSX AMD64	Linux AMD64	Linux x86	Linux ARM v7	AVR ATmega328	Native x86
Ackermann	121.79	72.48	67.40	24.34	0.02328	82.65
Array	185.25	75.66	63.20	26.57	0.02334	71.79
Fibonacci	98.91	56.13	54.59	23.53	0.02206	78.31
Heapsort	84.79	44.42	43.78	20.86	0.02183	62.72
Lists	25.31	21.67	20.11	10.93	-	60.83
Matrix	51.72	36.25	35.27	17.48	0.02170	67.39
Random	85.01	45.43	46.34	20.69	0.02068	62.77
Sieve	200.99	81.10	82.44	26.45	0.02387	83.44

Table 5. Benchmark performance (in MIPS) for each one of the platforms

In order to keep our emulation engine portable and compact, we did not implement interpreter optimizations, such as decoded instructions cache and threaded interpretation [30]. Even so, it was capable of emulating around tens of MIPS on modern x86 processors, which is on par with results reported by previous work [14].

The performance of the emulation engine on the AVR ATmega328 platform is about 2500 times worse than on the Linux x86 platform. This difference is due to three factors: a) clock frequency, b) number of instructions executed per cycle and c) average number of host instructions executed to emulate one guest instruction.

As we can see in Table 3, the processor of the Linux x86 run at 2.8 GHz, which is 175 times higher than the frequency of the processor at the AVR ATmega328 platform. Also, the processor at the Linux x86 platform is capable of executing up to 4 instructions per cycle[1] while the processor of the AVR ATmega328 platform can execute only one instruction per cycle. These two factors are strictly dependent on the hardware platform.

Table 6 shows the number of host per guest instructions executed to emulate the benchmark applications in each one of the platforms. Notice that the amount of host instructions required to emulate guest instructions on the AVR ATmega328 platform is about five times higher than the one on the Linux x86 platform.

	Linux AMD64	Linux x86	Linux ARM v7	AVR ATmega328
Ackermann	98.12	93.91	51.64	547.55
Array	109.03	102.55	53.34	545.06
Fibonacci	102.48	97.13	52.11	575.77
Heapsort	103.97	100.22	57.13	587.05
Lists	113.45	107.22	69.95	-
Matrix	104.95	100.92	62.40	721.55
Random	102.41	100.41	57.71	619.11
Sieve	95.53	92.76	52.87	532.58

Table 6. Host per Guest instructions, on average

The amount of host instructions required to emulate each guest instruction may depend on the emulation mechanism (see Dynamic Binary Translation vs Interpretation [22, 14]), the host compiler quality and the relationship between the guest ISA and the host ISA. Since we are using GNU avr-g++ 4.8.1 to compile the interpreter for the AVR ATmega328 platform and the GNU GCC 4.9.2 for the Linux x86 platform, we do not expect the host compiler quality to be a major player in this difference. Hence, the only explanation is the relationship between the guest and the host ISA. In fact, the AVR ATmega328 platform employs an 8-bit microcontroller, which has to execute multiple instructions to perform 32-bit data transfers and 32-bit pointer and arithmetic operations, the main kind of data and operations on the guest ISA.

On average, the number of host per guest instructions on the AMD64 platform is about the same as the one on the x86 platform. This is expected, since they are very similar ISAs. However, the curious fact comes from ARM. Even though x86 has more complex instructions, which enables it to execute the same computation with fewer instructions, our ARM Cortex A9 platform used, on average, almost half the number of host instructions to emulate a single guest instruction.

The *fetch-decode* loop is an important critical path in an interpreter because it is executed for every guest instruction. By carefully analyzing the binary produced by the x86 and ARM compilers, we found out that the x86 version uses 53 instructions in its fetch-decode loop alone, while ARM, 35: 11 to fetch, 10 to decode, 9 to advance the PC and 5 in the loop epilogue. In this case, the presence of the `ubfx` and `sbfx` ARMv6 instructions helped the ARM version perform fast field decoding with a single instruction,

[1] This number depends on several hardware and software factors and the average may be less than 4.

while x86 needed 3 instructions (`mov`, `and` and `shr`) to perform the same task. Finally, the small number of general-purpose registers also caused the fetch-decode loop of the x86 version to spill values to the stack, further increasing the number of instructions.

The design and implementation of our current emulation engine is focused on size and portability. However, we conjecture that it is possible to improve the COISA virtual platform performance on high-end platforms via Dynamic Binary Translation (DBT). In fact, our previous results [14] indicate that it is possible to implement a near-native performance DBT to emulate OpenISA code on x86 and ARM platforms. Also, the large number of host per guest instructions on the AVR ATmega328 platform suggests that there is still room to improve the interpreter on 8-bit resource constrained platforms. In this sense, in order to improve the performance on resource constrained platforms, we conclude it is important to investigate techniques to improve interpretation performance with little or no effect on the interpreter code and data footprint.

5. Conclusion

In this paper we proposed the COISA virtual platform and show that it is possible to emulate guest applications across a wide variety of host platforms, including platforms with severe resource constraints, such as the ATmega328 microcontroller.

Our experimental results show that a simple interpreter can be implemented using only a few bytes of memory and that this is enough to emulate general purpose programs. In fact, in our most restricted environment, our implementation used only 6 kB of flash memory and 348 bytes of RAM, leaving the remaining 1700 bytes of RAM to the guest applications. Moreover, we showed that the virtual platform can be built in a way such that its hardware-dependent portion can be easily ported to different platforms and yield the same results.

We also evaluate and discuss the performance of our virtual platform on 3 highly heterogeneous platforms and show that its portable interpreter is capable of emulating tens of millions of instructions per second on modern 32 and 64-bit x86 processors and on a 32-bit ARM processor. Even though this performance is on par with results reported by previous work [14], we conjecture that the system performance can be improved to native execution performance levels via DBT on high-end host platforms.

References

[1] A flyweight Python interpreter. https://wiki.python.org/moin/PyMite. Accessed: Jul. 2015.

[2] Apple II Emulator. https://courses.cit.cornell.edu/ee476/FinalProjects/s2007/bcr22/final%20webpage/final.html. Accessed: Jul. 2015.

[3] AVR NESEMU. https://courses.cit.cornell.edu/ee476/FinalProjects/s2009/bhp7/_teg25/bhp7/_teg25/index.html. Accessed: Jul. 2015.

[4] Commodore VIC-20 AVR Emulator. http://www.belanger.pwp.blueyonder.co.uk/Projects/Vic%20Emu/vicemu.htm. Accessed: Jul. 2015.

[5] Emulating a z80 computer. http://hackaday.com/2010/04/27/emulating-a-z80-computer-with-an-avr-chip/. Accessed: Jul. 2015.

[6] GCC Front Ends. https://gcc.gnu.org/frontends.html. Accessed: Jul. 2015.

[7] GNU Binutils. https://www.gnu.org/software/binutils/. Accessed: Jul. 2015.

[8] HaikuVM: a Java VM for ARDUINO and other micros using the leJOS runtime. http://haiku-vm.sourceforge.net/. Accessed: Jul. 2015.

[9] SimpleRTJ a small footprint Java VM for embedded and consumer devices. www.rtjcom.com/downfile.php?f=techpdf. Accessed: Jul. 2015.

[10] The NanoVM: Java for the AVR. http://www.harbaum.org/till/nanovm/index.shtml. Accessed: Jul. 2015.

[11] uJ: a Java VM for microcontrollers. http://goo.gl/VXdbS. Accessed: Jul. 2015.

[12] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, G. Ernst, E. Haussmann, S. Rührup, and Z. Uzmi. Optimized Java Binary and Virtual Machine for Tiny Motes. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2010.

[13] Atmel. 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash Datasheet. http://www.atmel.com/Images/doc8161.pdf, 2009. Accessed: Jul. 2015.

[14] R. Auler and E. Borin. OpenISA, freedom powered by efficient binary translation. In *Proceedings of the 8th AMAS-BT*, 2015.

[15] D. Bagley. Great Computer Language Shootout. http://dada.perl.it/shootout/. Accessed: Jul. 2015.

[16] M. Banzi, D. Cuartielles, T. Igoe, G. Martino, and D. Mellis. Arduino. http://www.arduino.cc/, 2005. Accessed: Jul. 2015.

[17] N. Bessis and C. Dobre. *Big Data and Internet of Things: A Roadmap for Smart Environments*. Studies in Computational Intelligence. Springer International Publishing, 2014.

[18] E. Borin and Y. Wu. Characterization of DBT overhead. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC '09)*, 2009.

[19] N. Brouwers, P. Corke, and K. Langendoen. A Java Compatible Virtual Machine for Wireless Sensor Nodes. SenSys '08. ACM, 2008.

[20] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov. Mote Runner: A Multi-language Virtual Machine for Small Embedded Devices. SENSORCOMM'09. IEEE, June 2009.

[21] A. L. Carvalho. Suporte para Execução de Máquinas Virtuais Nativas. Master's thesis, Institute of Computing, UNICAMP, 2015.

[22] D. Cesar, R. Auler, R. Dalibera, S. Rigo, E. Borin, and G. Araujo. Modeling virtual machines misprediction overhead. In *Proceedings of the IISWC*, 2013.

[23] S.B. Furber. *ARM System-on-chip Architecture*. Addison-Wesley, 2000.

[24] Intel. Intel Edison. http://www.intel.com/content/www/us/en/do-it-yourself/edison.html, 2014. Accessed: Jul. 2015.

[25] Joel Koshy and Raju Pandey. VMSTAR: Synthesizing Scalable Runtime Environments for Sensor Networks. SenSys '05. ACM, 2005.

[26] K. Krewell. Transmeta Gets More Efficeon. *Microprocessor Report*, 17(10), 2003.

[27] B. C. Lopes, R. Auler, L. Ramos, E. Borin, and R. Azevedo. SHRINK: Reducing the ISA Complexity via Instruction Recycling. ISCA 42. ACM, 2015.

[28] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual Machine Showdown: Stack Versus Registers. *ACM Trans. Archit. Code Optim.*, 4(4):2:1–2:36, January 2008.

[29] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. The squawk virtual machine: Java on the bare metal. VEE '06. ACM, 2006.

[30] J.E. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann, 2005.

[31] T. Suyama, Y. Kishino, and F. Naya. Abstracting IoT devices using virtual machine for wireless sensor nodes. In *WF-IoT*, pages 367–368, 2014.

[32] O. Vermesan and P. Friess. *Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems*. The River Publishers Series in Communications. River Publishers, 2013.