Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

ΑΝΑΠΤΥΞΗ ΕΙΚΟΝΙΚΗΣ ΜΗΧΑΝΗΣ ΓΙΑ ΕΤΕΡΟΓΕΝΗ ΕΝΣΩΜΑΤΩΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αντώνιος Κ. Τσίγκανος

Επιβλέπων : Δημήτριος Σούντρης
Επίκουρος Καθηγητής

Αθήνα, Μαρτιος 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

ΑΝΑΠΤΥΞΗ ΕΙΚΟΝΙΚΗΣ ΜΗΧΑΝΗΣ ΓΙΑ ΕΤΕΡΟΓΕΝΗ ΕΝΣΩΜΑΤΩΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αντώνιος Κ. Τσίγκανος

Επιβλέπων : Δημήτριος Σούντρης
Επίκουρος Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17η Μαρτίου 2014.

Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Γιώργος Οικονομάκος
Επίκουρος Καθηγητής	Καθηγητής	Λέκτορας

Αθήνα, Μάρτιος 2014

Αντώνιος Κ. Τσίγκανος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Τα ενσωματωμένα συστήματα στη σημερινή τους μορφή παρέχουν μεγάλο εύρος δυνατοτήτων και υψηλή απόδοση. Ωστόσο τους ζητείται να ικανοποιήσουν αντιτασσόμενους περιορισμούς στη συμπεριφορά τους, ενώ η αγορά απαιτεί σύντομους χρόνους ζωής. Λόγω αυτών των αναμεμειγμένων και αντιτασσόμενων περιορισμών, το σχεδιαστικό πρότυπο που έχει επικρατήσει είναι ο διαμοιρασμός της λειτουργικότητας του λογισμικού, πάνω σε ετερογενές υλικό. Ωστόσο αυτό το σχεδιαστικό πρότυπο δημιουργεί υψηλή πολυπλοκότητα, τόσο στον αρχικό σχεδιασμό, όσο και στη συντήρηση και αναβάθμιση του συστήματος.

Η αφαίρεση (abstraction) που παρέχουν οι εικονικές μηχανές μπορεί να απομονώσει τον προγραμματιστή απο αυτήν την πολυπλοκότητα. Σε αυτήν την εργασία προτείνουμε μια αρχιτεκτονική και παρέχουμε μια παραδειγματική υλοποίηση, την Portable Heterogeneous llvm Ir Virtual Machine (PHIVM), σε μια προσπάθεια να μελετηθούν οι επιδράσεις και οι απαιτήσεις των εικονικών μηχανών σε ετερογενή ενσωματωμένα συστήματα.

Η PHIVM επιτρέπει την φορητότητα εφαρμογών μεταξύ υπολογιστικών πυρήνων της ετερογενούς πλατφόρμας. Επίσης παρέχει τη δυνατότητα μετανάστευσης εφαρμογών κατα την εκτέλεση τους εντός του συστήματος, ενώ επιτρέπει ελαστικότητα στον σχεδιαστή του συστήματος στην χρήση ήδη υπάρχοντος σεναρίου διεπικοινωνίας. Αυτές οι δυνατότητες δεν απαιτούν αλλαγές στις φιλοξενούμενες εφαρμογές και είναι διαφανείς στην ανάπτυξη τους. Το πλαίσιο βιβλιοθηκών PHIVM βασίζεται στον LLVM, με αποτέλεσμα να υποστηρίζει πολλές γλώσσες προγραμματισμού ως είσοδο, όπως C/C++, Haskell κτλ. Είναι σχεδιασμένο με σκοπό την απλότητα και αποδοτικότητα, ώστε να είναι φορητό σε μεγάλο εύρος υπολογιστικών συστημάτων, παραμένοντας τροποποιήσιμο στις ανάγκες των σχεδιαστών.

National Technical University of Athens
School of Electrical & Computer Engineering
Division of Computer Science
Microprocessors and Digital Systems Lab

A virtual machine and runtime framework targeting Heterogeneous embedded systems

Antonios Tsigkanos

Diploma Thesis

Supervisor: Assist. Prof. **Dimitrios Soudris**

Abstract

Embedded systems of today, provide ample performance and capabilities. They need to satisfy opposing requirements in their behavior while the market demands short lifecycles. Due to these mixed and opposing requirements, the prevalent design pattern is becoming functionality partitioning in software, deployed on heterogeneous platforms. This design pattern though brings significant complexity in initial design as well as in maintenance and upgrade.

The inherent abstraction in Virtual Machines can isolate the programmer from much of this complexity. We propose in this work an architecure and provide a reference implementation, the Portable Heterogeneous llvm Ir Virtual Machine (PHIVM), in an effort to study the effects and requirements of Virtual Machines on heterogeneous embedded platforms.

PHIVM enables application portability across cores within the heterogeneous platform. It also provides for a task migration capability at runtime within the system, while allowing flexibility to the platform designer in using their already deployed inter-cpu communication scheme. These capabilities do not require any change in the VM-hosted application software and are transparent to its development. The PHIVM framework, being based on LLVM supports many input programming languages, such as C/C++, Haskell etc, and is designed to be as simple and efficient as it can be, to be easily portable in a wide variety of platforms while remaining modifiable to designers' needs.

Licensing

Acknowledgements

This work and the many lessons I have learned throughout it, would not have been possible without the invaluable help and support I have received by various members of the Microlab.

First, Assist. Prof. Dimitrios Soudris guided me in the work by providing invaluable insight and lessons that otherwise could only be accumulated by many years of experience. This study would not have been possible without his mentorship.

Additionally, Ph.D candidate Harry Sidiropoulos has given me time, incredible help and guidance in technical challenges, as well as an example of self discipline and perseverance when confronting engineering problems. I also thank Dr. Kostas Siozios for his considerable help throughout the course of this work.

Finally, I am indebted to Prof. Kiamal Pekmestzi, Assist. Prof. George Economakos as well as the MicroLab staff, for inspiring me through undergraduate courses to study embedded systems, by cultivating a culture of collaboration and exploration in the MicroLab.

Dedicated to to my family and my teachers.

Contents

List of Figures

Listings

Abbreviations

ES	**E**mbedded **S**ystem
OEM	**O**riginal **E**quipment **M**anufacturer
SoC	**S**ystem **o**n **C**hip
uC	Microcontroller
RISC	**R**educed **I**nstruction **S**et **C**omputer
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
DSP	**D**igital **S**ignal **P**rocessor
MAC	**M**ultiply **AC**cumulate
PLL	**P**hase **L**ocked **L**oop
NoC	**N**etwork **o**n **C**hip
MPSoC	**M**ulti **P**rocesssor **S**ystem **o**n **C**hip
ISA	**I**nstruction **S**et **A**rchitecture
VM	**V**irtual **M**achine
LLVM	**L**ow **L**evel **V**irtual **M**achine
LLI	**LL**VM **I**nterpreter
LLC	**LL**VM **C**ompiler
JIT	**J**ust **I**n **T**ime (Compiler)
API	**A**pplication **P**rogramming **I**nterface
ABI	**A**pplication **B**inary **I**nterface
vPC	**v**irtual **P**rogram **C**ounter
BTB	**B**ranch **T**arget **B**uffer
CLR	**C**ommon **L**anguage **R**untime
CIL	**C**ommon **I**ntermediate **L**anguage
MPI	**M**essage **P**assing **I**nterface
OSI	**O**pen **S**ystems **I**nterconnection (model)
NRE	**N**on **R**ecurring **E**ngineering

1 Introduction

This document describes the engineering diploma thesis project carried out by Antonios Tsigkanos, under the supervision of Assistant Professor Dimitrios Soudris, at the Microprocessors and Digital Systems Lab at the National Technological University of Athens.

This project involves the construction of a reference implementation of a virtualization framework, aiming to provide an abstraction over heterogeneous ISA Multiprocessor Systems on Chip, while providing a capability of seamless application migration at runtime, among differing ISA processing elements.

The document starts by outlining the motivation for this thesis, the technological context within which the research problem arises and a solution is considered. After the research questions are put, what follows is a description of relevant work and software frameworks as well as their components and properties that influence design and architectural decisions later on. Lengthier descriptions are included, namely of the frameworks the implementation of this work is based on, along with the rationale for their choice over antagonizing tools.

Along with a view of the solution space considering heterogeneous platforms later on, the possible and attempted solutions are outlined, while eventually arriving at the target requirements for a research prototype. The proposed architecture as well as the actual implementation are presented, noting the significant design choices as they are arise from architectural trade-offs and implementation challenges.

Finally, follows a discussion on the evaluation procedure of the architecture and a comparison with existing tools that could serve a similar purpose. A point by point analysis of features and associated trade-offs results in the design scenarios where the proposed solution is most appropriate.

This study concludes by revisiting the original motivation and research questions, with an overview of the work in light of the design implementation decisions, suggesting extensions and future implementations to research.

1.1 Embedded systems today

In this section, an outline is provided of the present state of embedded systems in the computing landscape. A comparison is drawn against general purpose computing, especially as it relates to their purpose in the consumer and industrial landscape and the way this influences design and development decisions and trade-offs. Considering the wide range of their application, we discuss the system constraints embedded systems are tasked to satisfy, as well as the design challenges involved in doing so. Considering these design challenges we highlight the heterogeneous hardware design pattern as a solution and it's associated complexities relating to development effort and time to market.

Embedded systems are defined as computing systems with a specific purpose and function within a larger electromechanical system, often with strict timing, power consumption, reliability and performance constraints. They are embedded inside a complete device usually including multiple sensors, actuators, network connectivity interfaces and other electrical, electronic and mechanical parts. In contrast to a general purpose computing system such as a consumer PC, they are designed to be highly flexible and configurable to meet systems engineers' needs in constructing function specific systems. In testament to their utility, an everyday modern consumer is surrounded by hundreds of embedded systems, performing multiple functions either actually facing the end consumer or within another system containing many symbiotic and collaborating embedded systems such as a smartphone or a car.

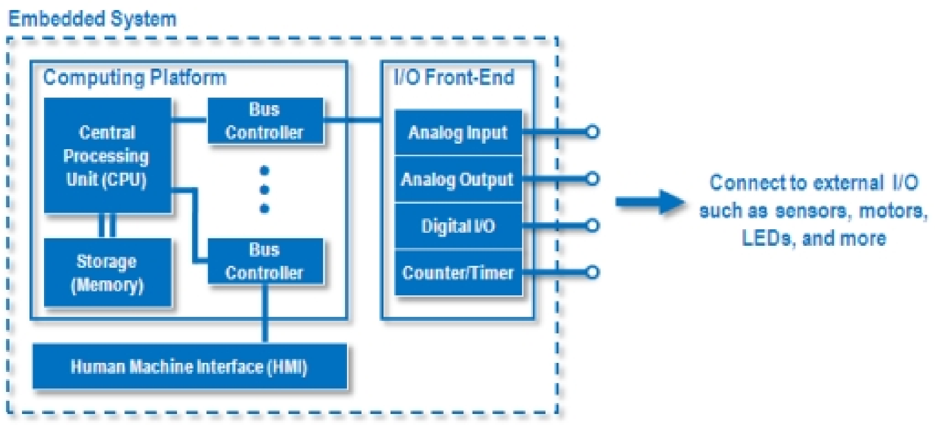


Figure 1: Embedded system example – block level abstraction(source: National Instruments)

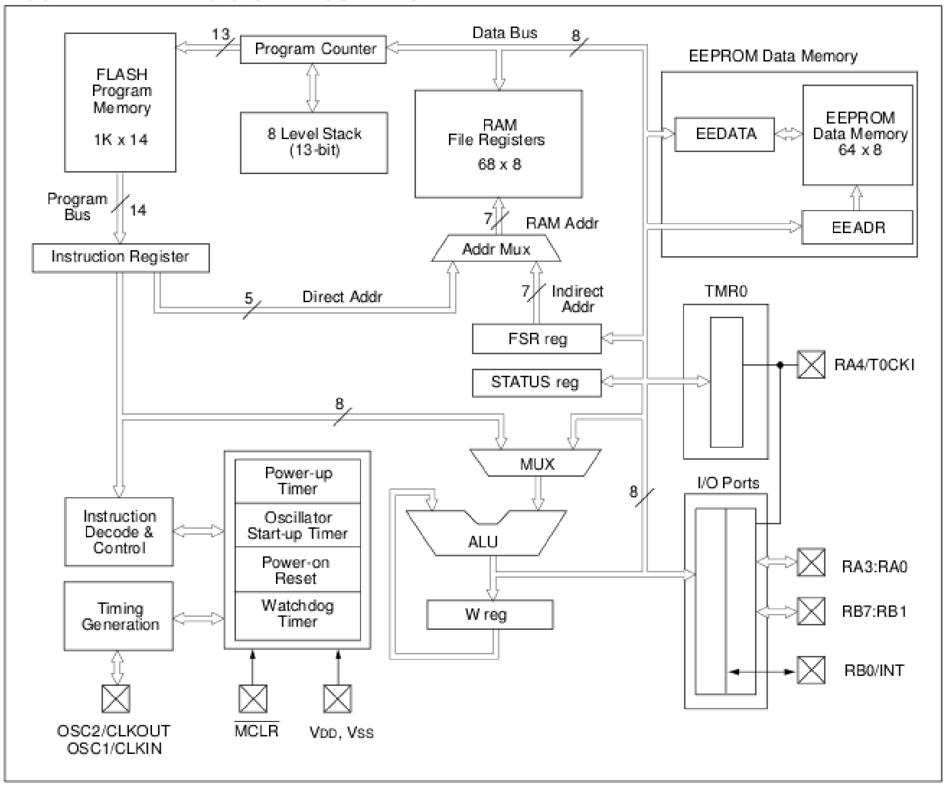


Figure 2: A typical microcontroller – Microchip PIC16F84A (source: Microchip)

The microcontroller – Continuing the definition of an embedded system in comparison against general purpose computing systems, we discuss their contributing parts. Most importantly but not exclusively, the principal component which is the central computing element of an embedded system is an microcontroller(uC). A microcontroller is a self-contained system with at least one processing element, memory (program and data memory) and (possibly) programmable input/output peripherals and can be used in an embedded system. In most cases they simple microcontrollers are used in systems with minimal requirements for memory and performance, without an operating system (baremetal), and low software complexity. Typically the processing element is a 8 or 32 bit word RISC Instruction Set Architecture operating at a frequency in the low MHz. Floating point operations are usually lacking except for DSP devices where floating point units may be accompanied by vector instructions and other special instructions such as multiply accumulate(MAC) etc.

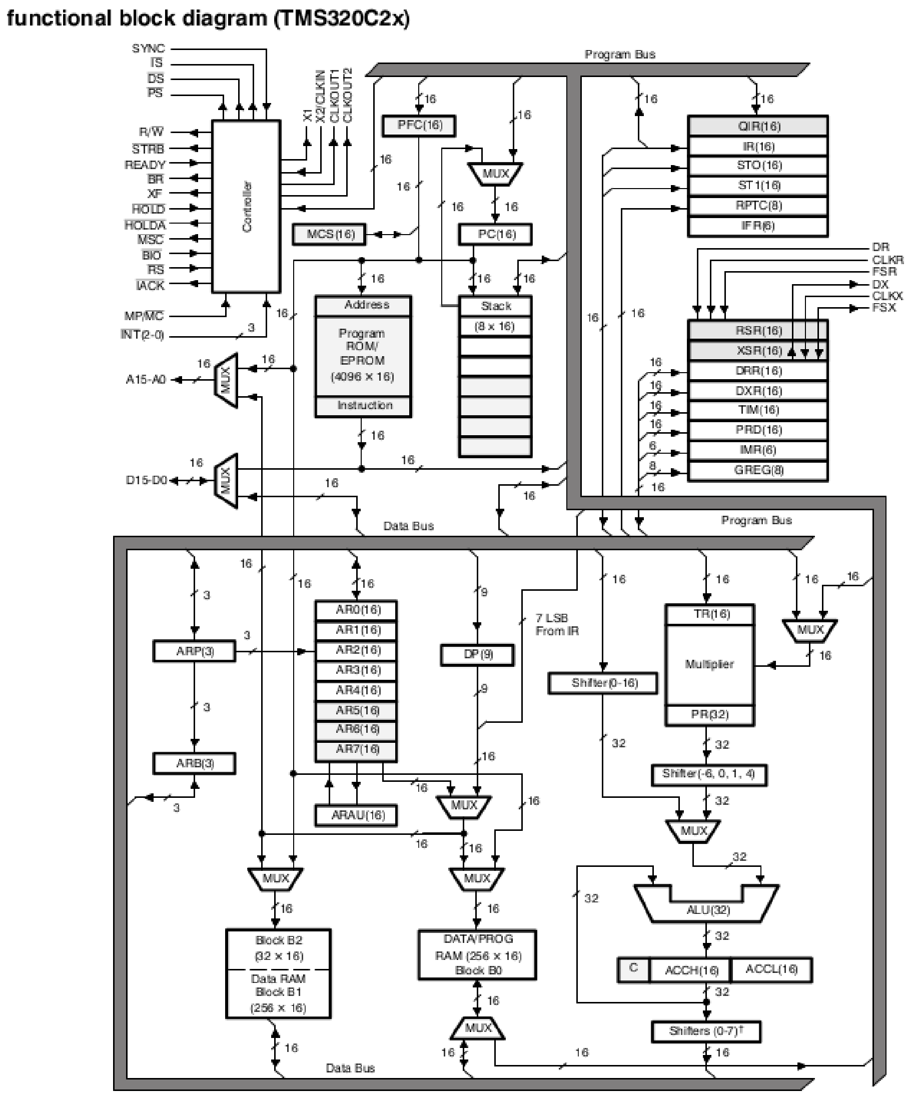


Figure 3: tms320 – a prolific TI DSP –(source: Texas Instruments)

The microcontrollers' memory systems input/output peripherals configuration as well as power management modes and available sleep states equally demand flexibility and configurability as much as the ISA. To address this need of widely varied configurations for different applications and target markets, we arrive at Systems on Chip.

Systems on Chip – Typically equipped with more processing power than a microcontroller, a System on Chip bundles processing elements with memory and many complex peripherals such as network interfaces into a single die. Depending on the target application a selection of at least some of the following components will comprise the System on Chip:

- A microcontroller, microprocessor or DSP core. Possibly a number of the same core might be replicated to build a Homogeneous Multiprocessor System on Chip. Alternatively a microprocessor and DSP might be paired to make a heterogeneous MPSoC.
- Memory blocks including a selection of ROM, RAM, EEPROM and flash memory.
- Timing sources including oscillators or configurable timing components such as phase-locked loops.
- Peripherals including watchdog timers, counter-timers, real-time clocks and power-on reset generators.
- Networking and inter-chip connectivity interfaces including industry standards such as USB, FireWire, Ethernet, USART, I2C, SPI.
- Wireless Networking such as Zigbee(802.15.4), bluetooth(802.15.1) or Wi-fi(802.11xx).
- Analog interfaces i.e. Analog to Digital Converters and Digital to Analog converters.
- Voltage regulators and power management circuits.

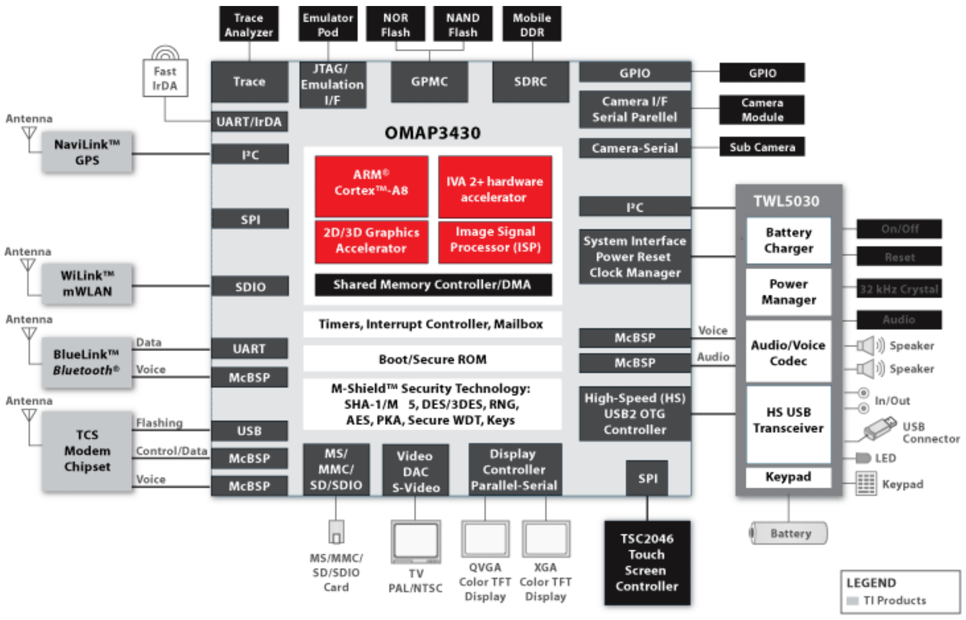


Figure 4: An example SoC block diagram – TI OMAP3430 – (source: Texas Instruments)

SoC Interconnect – The above blocks within the SoC are commonly connected by either a proprietary or open standard bus such as the AMBA bus from ARM or the Wishbone bus. Peripherals may also connect via DMA controllers directly between external interfaces to memory, increasing the data throughput of the SoC.

The purpose of a bus interconnect is to foster design reuse by alleviating SoC integration problems. This is accomplished by creating a common interface between IP cores which improves the portability and reliability of the system, and results in faster time-to-market for the end user.

Previously IP cores used non-standard interconnection schemes, that made them difficult to integrate requiring the creation of custom glue logic to connect each of the cores together. By adopting standard and open interconnection schemes, the cores and peripherals can be integrated more quickly and easily by the system designer. Bus interconnects are intended as general purpose interfaces, defining the standard data exchange between IP core modules without attempting to regulate the application specific functions of the module. They benefit systems designers and the industry as a whole by providing:

- a reliable System-on-Chip integration solution.
- common interface specification to facilitate structured design methodologies on large project teams.
- a generalized and flexible integration solution that can be easily tailored to a specific application.
- a variety of bus cycles and data path widths to solve various system problems.
- cross compatibility allowing IP cores to be designed by a variety of vendors.

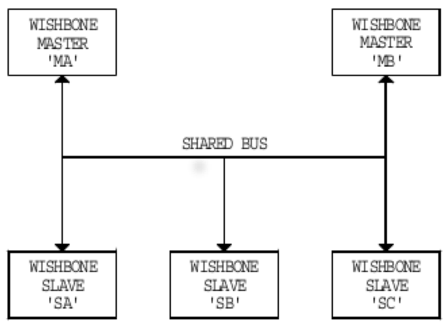


Figure 5: Shared SoC Bus example –(source: [3])

Increasingly however scalability becomes a concern as the number of processing elements and peripherals rapidly increases as well as advances in silicon technology rebalance the interconnect trade-offs. Inherently, busses do not decouple the activities generally classified as transaction, transport and physical layer behaviors. This is the key reason they cannot adapt to changes in the system architecture or take advantage of the rapid advances in silicon process technology. Consequently, changes to the bus physical implementation can have severe ripple effects on the implementations of higher-level bus behaviors; thus causing systems based on bus architectures to not closely follow process evolution nor system architecture evolution.

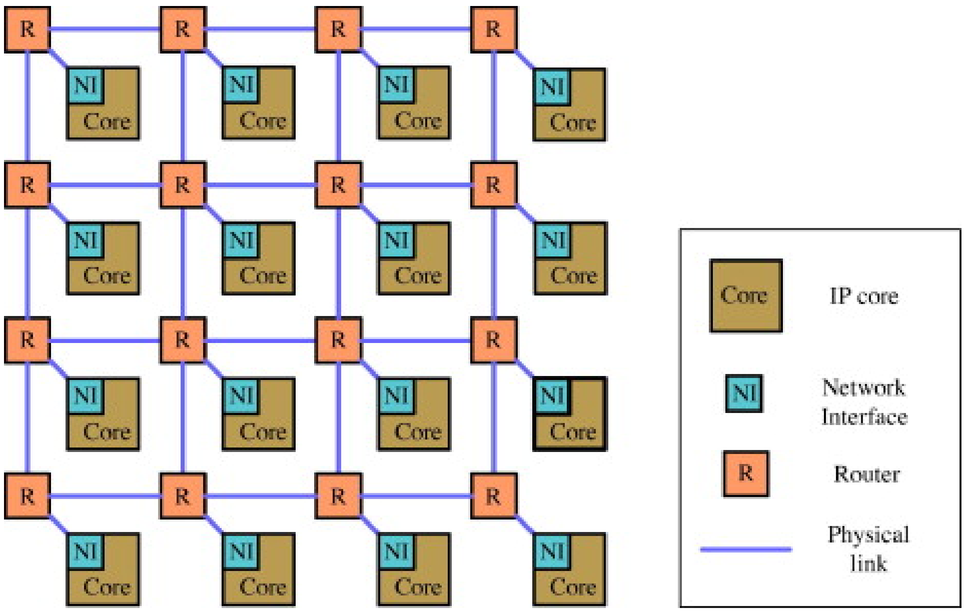


Figure 6: SoC Network on Chip –(source: IST/INESC-ID)

In the data communications space, Local and Wide Area Networks have successfully dealt with similar problems by employing the OSI model of a layered architecture. The decoupling of communication layers using the OSI model has successfully driven commercial network architectures, and enabled networks to follow very closely both physical layer evolutions and transaction level evolutions. This has introduced flexibility at the application level, while maintaining upward compatibility.

Following the same trend, networks have started to replace busses in much smaller systems: PCI-Express, a network-on-a board replaces the PCI board-level bus. Replacement of SoC busses by NoCs are slowly following the same path, when the economics are proving that NoCs are:

- Reducing SoC manufacturing cost
- Enabling scalability in system architecture
- Increasing SoC performance
- Reducing SoC time to market and NRE
- Reducing SoC design risk

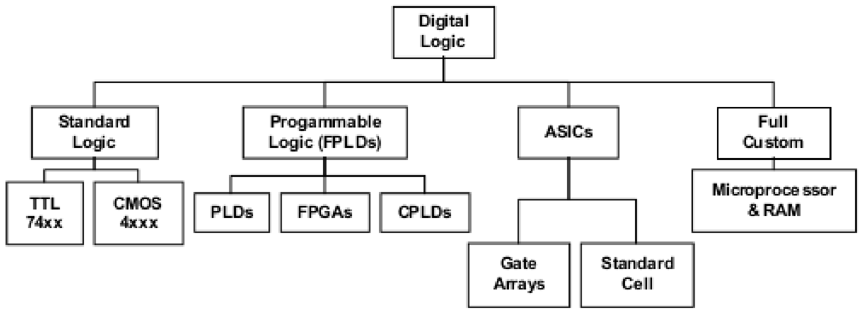


Figure 7: Logic Devices tree

The rise of programmable logic devices – In the last decade, higher densities, higher clock speed, and cost advantages have enabled the use of programmable logic devices in a wider variety of designs and applications. CPLDs and FPGAs are the highest density and most advanced programmable logic devices for which designs typically only require several weeks of engineering effort instead of months.

On the opposite side of this trade-off, ASICs and full custom designs provide faster clock speeds than CPLDs or FPGAs since they are hardwired and do not have programmable interconnect delays. Since ASICs and full custom designs do not require programmable interconnect circuitry they use less chip area, less power and have a lower per unit manufacturing cost in large volumes. This performance and low cost though is traded for by much higher initial engineering and setup costs for ASICs and full custom designs. For all but the most time critical design applications, CPLDs and FPGAs have adequate speed with maximum clock rates typically around 400MHz; however, clock rates up to 1GHz have been achieved on new generation FPGAs and many have a few high-speed 1-10 GHz output pins. As seen in the following abstract visualization, the design decision between ASICs and FPGAs boils down to design performance against NRE and time to market.

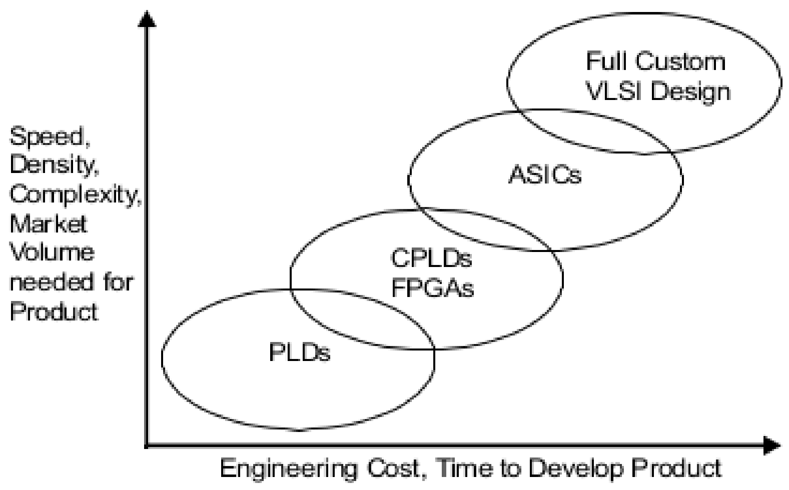

Speed, Density, Complexity, Market Volume needed for Product
Engineering Cost, Time to Develop Product
Full Custom VLSI Design
ASICs
CPLDs FPGAs
PLDs

Figure 8: Device technology tradeoffs – (source: Rapid prototyping of digital systems Hamblen, Hall, Furman)

Internally, CPLDs and FPGAs typically contain multiple copies of a basic programmable logic element or cell. The logic element can implement a network of several logic gates that are then fed into 1 or 2 flip-flops. Logic elements are arranged in a matrix on the chip. To perform complex operations, logic elements are automatically connected to other logic elements on the chip using a programmable interconnection network which is also contained within the FPGA. The interconnection network used to connect the logic elements contains row and column chip-wide interconnects. In addition, the interconnection network often contains shorter and faster programmable interconnects limited only to neighboring logic elements.

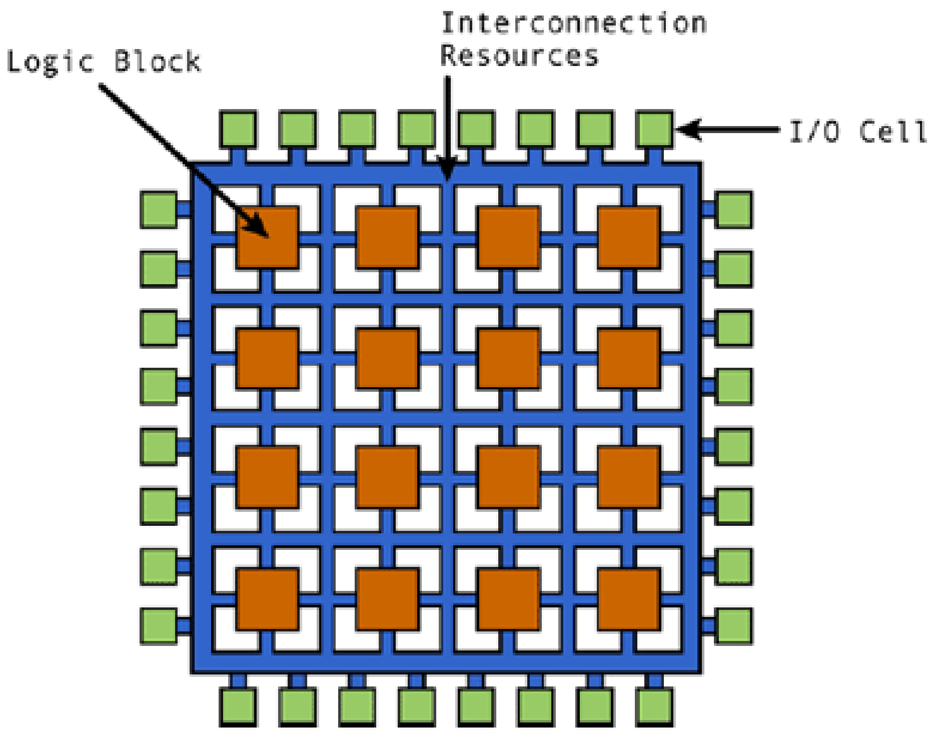


Figure 9: Abstract view of typical FPGA – (source: eetimes.com)

Contributing in FPGAs versatility most devices include a host of "hard" resources dispersed in the "sea of gates" and available to the design, most notably:

- Block RAMs adding up to several megabytes
- Digital Signal Processing elements such as the below DSP48 element.
- Networking elements supporting ethernet pci-e etc.
- Clock management elements such as PLLs.

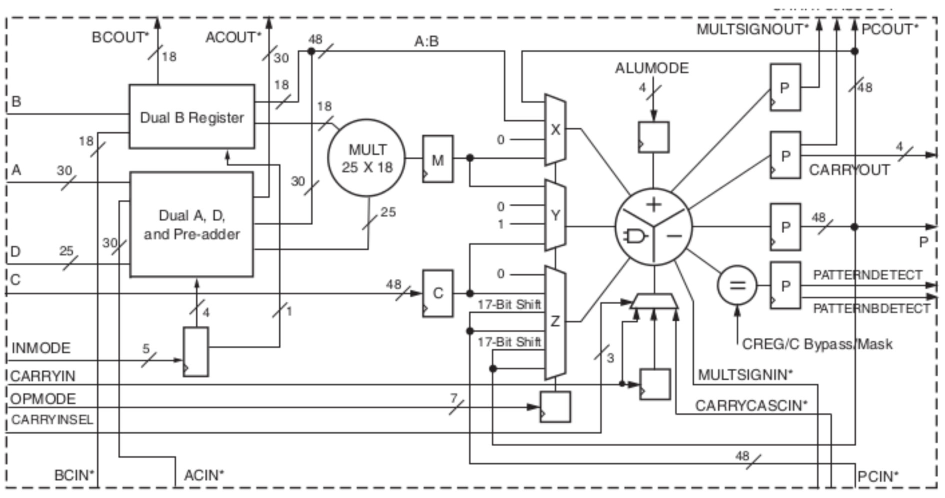


Figure 10: A Xilinx DSP48E1 included in Virtex-6 devices – (source: Xilinx)

Elements such as the above can usually be inferred by the design tools from the hardware description and automatically included in the resulting hardware design allowing great performance and power consumption benefits as well as relaxing pressure on the interconnection resources.

Systems on Programmable Chip – In parallel to high integration for Systems on Chip and scalability allowed by advanced interconnects, silicon process technology has enabled the ultimate in design flexibility and configurability: Systems on Programmable Chip. The most interesting of the programmable logic devices, the Field Programmable Gate Array (FPGA) has been riding Moore's law in the last decade, with devices exploding in capabilities, size and clock speed.

High speed and large capacity FPGAs have enabled design engineers to implement completely integrated within programmable logic, complete systems. In this form SoPC include microprocessors, memory hierarchies and all the peripherals the requirements demand in the reconfigurable fabric of the FPGA potentially also utilizing the "hard" elements mentioned above. This design flexibility has allowed a tight coupling of hardware and software, which increases performance by focusing resources on the target application. The low clock speed of the device is thus compensated for, wasting no time or resources by the hardware being general purpose.

Following that same trend, device manufacturers have responded by adding complete "hard" microprocessors inside the reconfigurable fabric. This avoids the design and performance overhead of describing the processor hardware and implementing it in reconfigurable logic. Resulting in significant benefits in time to market, performance and power efficiency these devices have seen mass adoption.

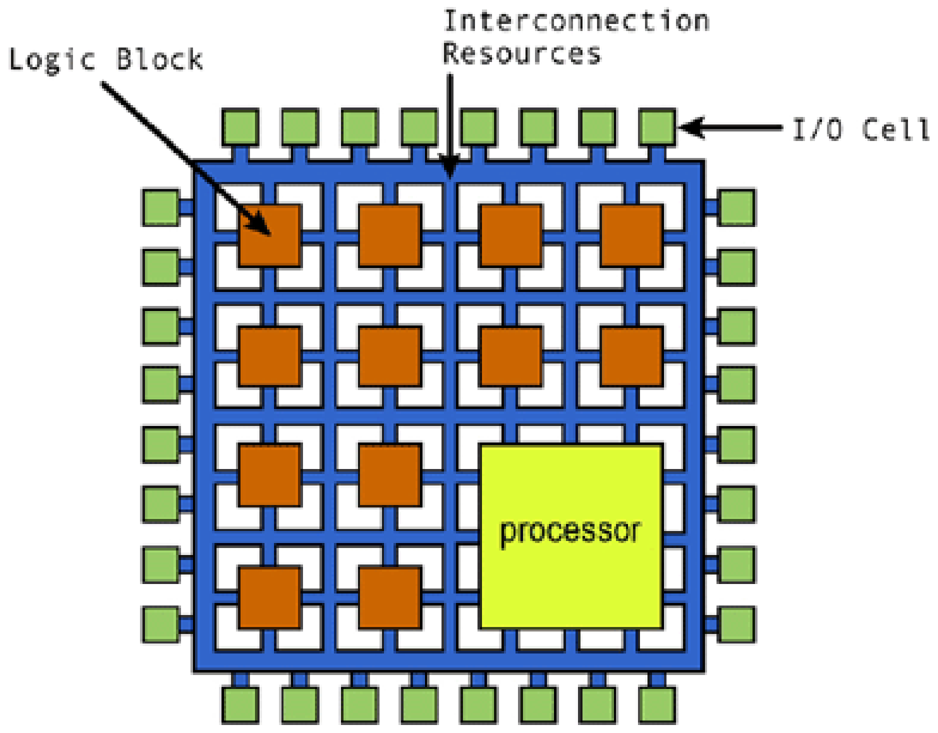


Figure 11: A hard microprocessor surrounded by reconfigurable fabric – (source: eetimes.com)

The inverse of this trend, placing configurable logic around a processor rather than placing the processor inside a reconfigurable chip, is also emerging. An example of this, the Cypress PSoC, uses PLDs connected to the SoC bus along with the usual peripherals.

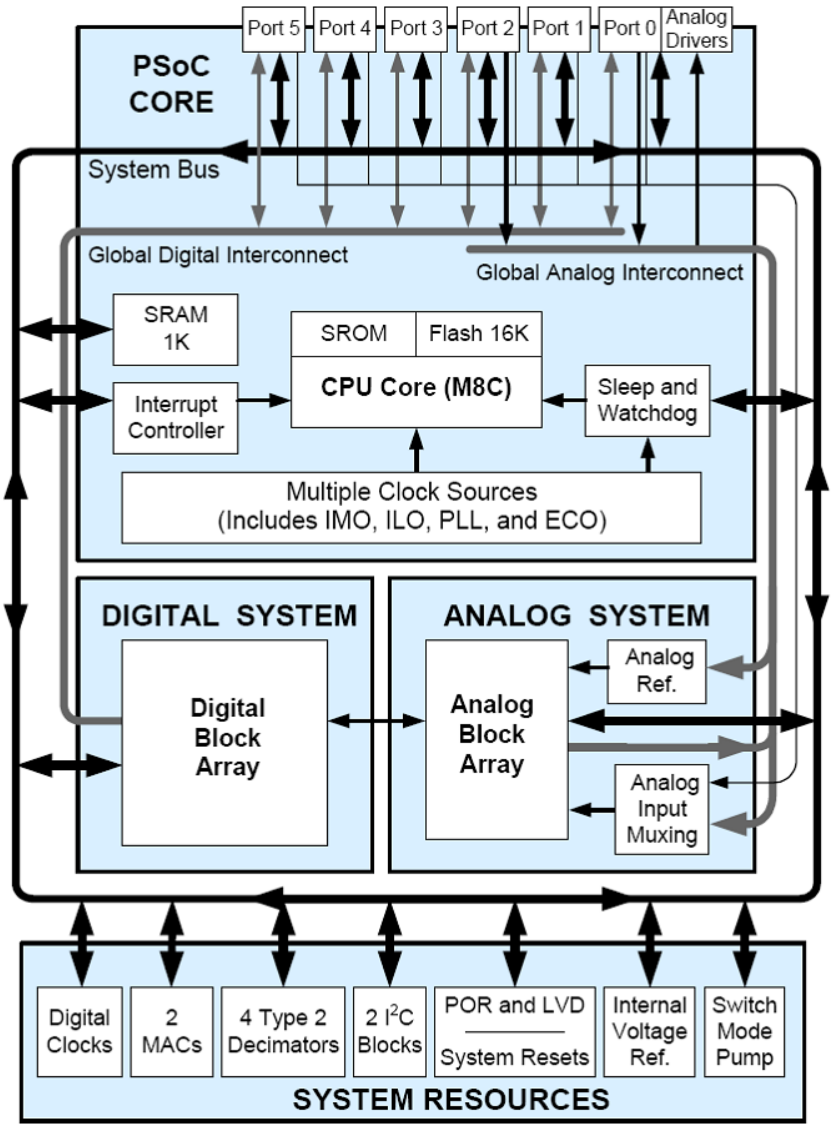


Figure 12: A Cypress PSoC with configurable peripherals– (source: Cypress Semi.

FPGA partial reconfiguration Adding to the already impressive list of reconfigurable logic's benefits, partial reconfiguration is the latest available feature. It is the ability to reconfigure only part of the FPGA, even at run-time while other parts of the logic continue operations uninterrupted. Two basic types of partial reconfiguration are usually supported: module-based and difference-based. Module-based partial reconfiguration uses modular design blocks to reconfigure large blocks of logic. The distinct portions of the design to be reconfigured are known as reconfigurable modules. Because specific properties and specific layout criteria must be met with respect to a reconfigurable module, any FPGA design intending to use partial reconfiguration must be planned and laid out with that in mind. Difference-based partial reconfiguration is a method of making small changes in an FPGA design, such as changing I/O standards, LUT content, and block RAM content.

Partial reconfiguration can also be performed by implementing a basic controller to manage the reconfiguration of the FPGA. This could be in the form of an external processor, or one programmed within the same FPGA in the static portion of the design. Frequently, the parts of the functionality that can be accelerated through the use of reconfigurable hardware are too many or complex to be loaded simultaneously onto the available logic. In these cases, it is beneficial to be able to swap different configurations in and out of the reconfigurable hardware as they are needed during program execution.

This concept is known as run-time reconfiguration. Run-time reconfiguration is based on the concept of virtual hardware, similarly to virtual memory. Here, the physical hardware is much smaller than the sum of the resources required by each of the configurations. Therefore, instead of reducing the number of configurations that are mapped, the reconfiguration processor swaps them in and out of the actual hardware as they are needed. Because run-time reconfiguration allows more sections of an application to be mapped into hardware than can be fit in a non-run-time reconfigurable system, a greater portion of the program can be accelerated. This provides great potential for an overall improvement in performance and full utilization of programmable logic.

1.2 Heterogeneous ISA multiprocessing systems

From the state of embedded systems outlined in the previous section a few trends are becoming apparent:

- Embedded systems being application specific and optimized by definition, need flexibility and configurability in the devices and platforms on which they are based.
- This demand for flexibility has spread from the software into the lower levels of hardware.
- Systems are increasingly multiprocessing.
- The capabilities offered by programmable devices enable function specific co-processors along with general purpose processing.
- Increased power efficiency requirements are driving design tradeoffs.
- Timing and performance requirements are driving partitioning of tasks by priority unto separate parts of the device.
- Mixed constraints are increasing system complexity.

These trends and new capabilities are inevitably leading to highly heterogeneous multiprocessing systems to meet requirements. Heterogeneity is becoming prevalent in most high performance embedded platforms, carrying combinations of a main processor and highly optimized co-processors, or pairs of processors of a different ISA. Microprocessor ISAs have vastly different performance profiles across different applications making the selection of an appropriate ISA an important early design decision.

Reconfigurable chips large and fast enough to host multiple processors, as well as the availability with cheap licensing or completely free, of IP cores for multiple ISAs increasingly push heterogeneity. At the time of writing a considerable number of processors are openly licensed or relatively easy to license. Most importantly these very common ISA families are well understood, mature and well documented. Their popularity results in the availability of mature and robust development tools and end-user software for most applications.

- openSPARC (GPL)
- LEON (LGPL/GPL)
- ARM Holdings ARMxx family (closed source but easily licensed)
- openRISC (LGPL)
- MIPS families
- Picoblaze (free to use but closed license: Xilinx)

- NIOS II (free to use but closed license: Altera)

These are the most common and widely used families, each having versions optimized for various functions increasing utility while maintaining mainline compatibility for software tools and applications. Apparent from this list is that design engineers have ample choices for system architecture. This design freedom is coupled with an equal availability of system interconnects in busses and networks on chip. This results in heterogeneous multiprocessing systems connected with easy to deploy interconnects, on reconfigurable platforms to leverage considerable benefits in meeting constraints.

It can be argued that the key problem in embedded system design that does not exist in general purpose computing systems, is that of mixed constraints. Generally an embedded system will be called to strictly satisfy at least some of the following constraints:

- Timing
- Reliability
- Performance
- Power efficiency

This of course is the bane of engineers' lives because the solution to these constraints, generally live on opposite sides of tradeoffs. These tradeoffs are sometimes imposed by software design patterns but more often than not by the properties of the ISA. Constrained based partitioning of functionality and subsystem purpose, is increasingly used to solve this.

Consider the most canonical example of this in systems where telecommand and telemetry functionality must coexist. Telecommand must satisfy hard timing and reliability so that the system will be reliably managed. On the other hand telemetry is computationally intensive due to data processing or more complex operations, but it is not operationally critical and can gracefully fail or be suspended for power consumption purposes. Typically such a system will employ a heterogeneous layout. Reliability and timing will be satisfied by a small, power efficient master processor. The master would control a data acquisition and processing unit of an ISA appropriate for high performance with digital signal processing capabilities. The data processing element can even be scaled to multiple processors without compromising reliability or introducing complexity.

This heterogeneous ISA design pattern enables partitioning of functionality unto the hardware most capable of performing it. By assigning system requirements to discreet elements of the hardware platform, each hardware element can be independently optimized and developed along with it's corresponding software. Thus it becomes easier to meet constraints while simplifying the hardware-software codesign process.

This design pattern employed by systems engineers has not gone unnoticed by silicon OEMs which have responded with heterogeneous devices and platforms. Most notable is the ARM Holdings big.LITTLE architecture, although only "slightly Heterogeneous" it uses cores of the same ISA family to distribute computing load for power savings.

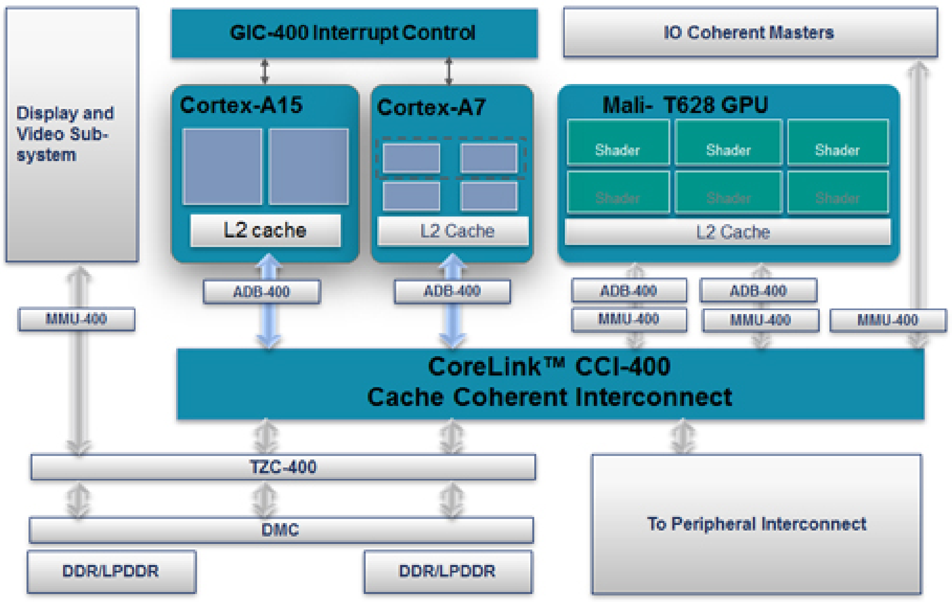


Figure 13: ARM big.LITTLE – (source: ARM)

The big.LITTLE architecture takes advantage of the fact that the usage pattern for consumer computing is dynamic: Periods of high processing intensity tasks, alternate with typically longer periods of low processing intensity tasks such as waiting for user input in end-user facing applications. A set of high performance and high efficiency CPU clusters are connected through a cache coherent interconnect fabric such as the ARM CoreLink. The processors appear as a multicore cpu to the operating system. User space software on a big.LITTLE SoC is identical to the software that would run on a standard SMP processor.

Scheduling to the right processor is accomplished by a kernel space patch. This makes the Operating System aware of the big and LITTLE cores, and gives the ability to schedule individual threads of execution on the appropriate processor based on dynamic run-time behavior. The software also keeps track of load history for each thread that runs, and uses the history to anticipate the performance needs of a thread the next time it runs.

big.LITTLE software automatically handles the allocation of threads of execution to the appropriate CPU.

There are three different modes of operation for the software, all based on the same hardware description. In the global task scheduling model of software, the operating system is directly aware of the high performance and high efficiency cpus in the system. In that mode of operation, the software dynamically allocates each thread based on the performance required. The software uses a load history to remember previous performance demands, and a dynamic load tracker to adapt to runtime performance that may differ from the load history. The software reacts quickly to changes in load, and can move work to the big or LITTLE cpu cluster in less time than an SMP load balancing action, invisibly in the background.

Although this abstraction and obscurity of underlying heterogeneity is ideal, it is only possible in this case due to use of the same family ISAs in the processing elements. A highly optimized architecture would require in turn optimized ISAs possibly in-house developed or modified for the application at hand. The big.LITTLE architecture in this regard is not representative of extreme heterogeneity since it targets consumer applications and not highly specialized ES. To properly leverage the benefits of heterogeneous architectures, an abstraction is needed to simplify development and deployment, for which we look into Virtual Machines.

1.3 Virtual machines and Managed runtime environments

Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning , partial or complete machine simulation, emulation, time-sharing, and others. Virtualization technologies have wide applications over a range of areas such as code mobility, secure computing platforms, supporting multiple operating systems, kernel debugging and development, system migration, etc, resulting in widespread usage. They tend to vary widely in their levels of abstraction they operate at and the underlying architecture. We can identify the following abstraction levels on which VMs operate: instruction set level, hardware abstraction layer (HAL) level, operating system level, library level and application level virtual machines.

In computing, a virtual environment is perceived the same as that of a real environment by application programs, though the underlying mechanisms are formally different. More often than not, the virtual environment presents a misleading image of a machine that has more or less capability compared to the actual physical machine underneath for various reasons.

A typical computing software stack already uses many such abstraction technologies. One prolific example is the virtual memory implementation in any modern operating system that lets a process use memory typically much more than the amount of physical memory its computer has to offer. Similarly, multitasking can be thought of as another example where a single cpu is partitioned in a time shared way to present some sort of a virtual cpu to each task. There are quite a few of examples in today's world that exploit such methods. The umbrella of technologies that help build such virtualized objects can be said to achieve tasks with one common methodology, virtualization. With the increase in applications of virtualization concepts across a wide range of areas in computer engineering, the width of the definition is ever increasing. For this discussion consider the following generalized definition:

"Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and many others"[5]. A virtualization layer, as follows provides infrastructure support using the low level resources to create virtual machines that are independent of and isolated from each other. There are many reasons for how virtualization can be useful in practical scenarios, a few of which are the following:

- Application consolidation: A legacy application might require newer hardware and/or operating systems. Fulfillment of the need of such legacy applications could be served well by virtualizing the newer hardware and providing its access to others.

- Code Portability: It is generally true that a program built within an OS and library environment is not easily ported to another. This creates a compatibility problem across operating systems and execution platforms. Virtualization can be used to develop programs for a specific virtual machine transferring portability responsibilities to the VM rather than the application software.
- Software Migration: Eases the migration of software and thus helps mobility.
- Appliances: Lets one package an application with the related operating environment as an appliance.
- Sandboxing: Virtual machines are useful to provide secure, isolated environments for running foreign or less-trusted applications. Virtualization technology can, thus, help build secure computing platforms.
- Multiple execution environments: Virtualization can be used to create multiple execution environments and can increase the QoS by guaranteeing a specified amount of resources.
- Multiple simultaneous OS: It can provide the facility of having multiple simultaneous operating systems that can run many different kind of applications.
- Debugging: It can help debug complicated software such as an operating system or a device driver by letting the user execute them on an emulated PC with full software controls.
- Testing-Quality Assurance: Helps produce arbitrary test scenarios that are hard to produce in reality and thus eases the testing of software.

Conceptually a virtual machine represents an operating environment for a set of user-level applications, which includes libraries, system call interface/service, system configurations, daemon processes, and file system state. There can be several levels of abstraction where virtualization can take place: instruction set level, hardware abstraction layer (HAL), OS level (system call interface), user-level library interface, or in the application level. Whatever may be the level of abstraction, the general phenomenon still remains the same; it partitions the low level resources using some novel techniques to map to multiple higher level VMs transparently.

Emulation is the technique of interpreting the instructions completely in software. For example, an ARM emulator on an X86 processor can execute any ARM application, thus giving the illusion to the application as if it is on a real ARM processor. To achieve this, however, an emulator would have to be able to translate the hosted ARM ISA to the hosts X86 ISA.

The functionality and abstraction level of a HAL level virtual machine lies between a real machine and an emulator. A virtual machine is an environment created by a VM manager (VMM), which is the virtualization software lying between the bare hardware and the OS and gives the OS a virtualized view of all the hardware. A VMM can create multiple VMs on a single machine. While an emulator provides a complete layer between the operating system or applications and the hardware, a VMM manages one or more VMs where every VM provides facilities to an OS or application to believe as if it runs in a normal environment and directly on the hardware.

Virtualization at the instruction set architecture level is implemented by emulating an instruction set architecture completely in software. An emulator tries to execute instructions issued by the hosted machine by translating them to a set of native instructions and then executing them on the the available hardware. These instructions would include those typical of a processor, and the I/O specific instructions for the devices. For an emulator to successfully emulate a real computer, it has to be able to emulate everything that a real computer does that includes reading ROM chips, rebooting, switching it on, etc.

Although this virtual machine architecture works fine in terms of simplicity and robustness, it has its own pros and cons. On the positive side, the architecture provides ease of implementation while dealing with multiple platforms. As the emulator works by translating instructions from the guest platform to instructions of the host platform, it accommodates easily when the guest platforms architecture changes as long as there exists a way of accomplishing the same task through instructions available on the host platform. In this way, it does not enforce a stringent binding between the guest and the host platforms. However, the architectural portability comes at a price of performance. Since every instruction issued by the emulated computer needs to be interpreted in software, the performance penalty involved is significant.

The JVM – In the context of managed runtime environments for embedded systems we are interested in bytecode VMs. The most notable of which is of course the JVM. The JVM is a virtual machine that runs Java byte code. This code is usually generated by Java compilers, although the JVM has also been targeted by compilers of other languages which speaks to it's maturity. Programs intended to run on a JVM must be compiled into a standardized portable binary format comprised of Java byte code, which typically comes in the form of .class files.

This binary is then executed by the JVM runtime which carries out emulation of the JVM instruction set by interpreting it or by applying a just-in-time Compiler (JIT). The JVM, in addition to providing the instruction set interpreter, also provides the operating environment for the Java byte codes. Thus the Java platform is a combination of a virtual machine along with an operating environment (JRE). The virtual machine is eventually implemented in some native language and can afford to be more flexible than traditional machines. It adds some extra computation to add features like byte code verification, structured exception handling, garbage collection and so on. Being in the application layer, it has much more control over these implementation than system implementations. The JVM is a stack-based architecture and supports threads. Each thread has its own program counter and virtual register set (registers supported by the Virtual Machine). These instructions, at runtime, are mapped to a set of real instructions that are to be executed natively. Code verification also ensures that arbitrary bit patterns cannot get used as an address. Memory protection is achieved without the need for a memory management unit. Thus, JVM is an efficient way of getting memory protection on simple silicon that has no MMU.

The JVM supports instructions like load/store, arithmetic, type conversion, object creation/manipulation, push/pop, branches, call/ret, and exception throws. However, more than the emulation of the byte code is the complication involved in getting a compatible and efficient implementation of the map of Java core API to the host OS. The virtual hardware of the Java Virtual Machine can be divided into four basic parts: the registers, the stack, the garbage-collected heap, and the method area. These parts are abstract, just like the machine they compose, but they must exist in some form in every JVM implementation. JVM supports addresses upto 4GB of memory with its 32-bit addressing scheme and uses 32-bit virtual registers. Depending on the particular JVM implementation, the stack, garbage-collected heap, and the method area reside at some well-defined places within this memory. JVM supports a small number of primitive data types: byte (8 bits), short (16 bits), int (32 bits), long (64 bits), float (32 bits), double (64 bits), char (16 bits), and object handle (32 bits).

Apart from the program counter, it uses three registers to manage the stack: optop register, frame register, and vars register. These point to various places within the stack of the executing method. Method area is similar to the text area in an x86 machine; it contains the byte code to be executed and the program counter always points to some byte in this area. The program counter is similar to PC and advances as execution proceeds. The stack is used to store the parameters and results of the methods, and to keep the state of each method invocation. The vars register point to the local variables section containing all the local variables in the method. Frame register points to the execution environment within the stack that maintains the operations of the stack. Finally, the optop register points to the top of the stack where the operands and results are placed. Such a virtual machine architecture allows very fine-grained control over the actions that code within the machine is permitted to take. This allows safe execution of untrusted code from remote sources, a model used most famously by Java applets. Security, sandboxing, easy debugging, platform-independence are a few very important features of such a virtualization setup. Since all the hardware devices are below the JVM layer, it has access to everything in the system and virtually do everything that a normal application can do.

In conclusion to this short review of VMs, their most important features in the context of the challenges of embedded systems are:

- They are proved and well studied in general purpose computing.
- They are a mature technology, their design and implementation trade-offs being well studied.
- They can provide a robust abstraction over the underlying ISA and hardware complexity
- They enable higher agility in software development.
- They can enable wide support of languages and underlying platforms

1.4 Motivation

It has become clear that new capabilities in embedded systems design are providing great potential benefits. These benefits though, introduce significant design complexity and require great investment in development effort.

As we argued in the previous section, the heterogeneous design pattern, enabled by capable programmable chips, can be amplified by features such as advanced interconnect schemes and flexible memory hierarchies. These combinations can prove extremely powerful in meeting constraints and increasing flexibility in the final system. Alas these benefits come at a high initial cost in the design and development stage (high NRE costs) and introduce considerable system complexity in the lifetime of the system.

Uniquely to heterogeneous systems, software development tools and the resulting code is very hard to manage. Cross-compilation toolchains, the associated libraries for the target platforms and operating systems are more difficult to manage and maintain throughout the development and platform lifetime. Not only are these notoriously difficult to setup, port and maintain on custom ISAs and platforms but on a hardware platform, developed and refined in tandem with it's software, it is practically impossible. Yet hardware platform evolution throughout project lifetime, is crucial to enabling software-hardware co-design.

As a result, especially in such heterogeneous systems initial design choices lock the platform due to development tools libraries' setup and application software partitioning on processing elements. This lock in the initial design choices negates most of the benefits of a heterogeneous programmable logic platform in the first place! Moreover if initial design choices were ill informed one may end up with an underperforming complex system with poorly chosen hardware and locked by existing software, too expensive to re-develop.

Recognizing the great potential of heterogeneous systems we seek a solution to manage the complexity they introduce. From such a solution we would like:

- To reduce development effort and avoid locking the system to a possibly non-optimal platform due to lack of information early in the design process.
- To allow leveraging heterogeneity for application functionality partitioning on diverse processing elements. This assignment of tasks to processing elements would be static but ideally we would like to be able to rearrange the allocation to processing elements.
- To be able to leverage runtime reconfiguration, to change platform features at runtime, without significant impact on software.

- To be able to increase the processing elements as needed and dynamically alocate tasks to them. Similarly to remove processing elements, possibly for power efficiency, by moving their uncompleted tasks back to fewer processors.

To recap we would like to be able to decouple software and the underlying hardware. This both for platform design exploration and research in early stages of system development and at runtime for load balancing and meeting different requirements in application characteristics.

Faced with increasing complexity in heterogeneous systems, we are looking for a solid but flexible and robust abstraction to separate efforts in hardware and software development but also enable more highly abstracted design decisions. This abstraction layer should provide portability across candidate ISAs for heterogeneous systems. Should aid in exploring possible configurations of the platform without hindering development of application software in parallel. It should allow for application migration across processing elements, during development for deciding on the partitioning of tasks on the platform, but also during runtime for dynamically adapting to requirements.

In general purpose computing, similar problems have been solved with Virtual Machines and managed runtime environments. We propose a Virtualization solution for heterogeneous embedded systems it's purpose being to ease software-hardware codesign, reduce development effort for portability and code mobility and allow the exploration of dynamic task allocation on processing elements by providing a task migration capability.

1.5 A hypothetical embedded system deployment scenario

In this subsection we consider a generalized embedded platform that fits to many real world applications. This system has time and reliability sensitive constraints for managing itself and it's interfaces, a networking element through which it is possibly controlled. It's main function though is computationally intensive, "power hungry" but non critical.

This generalization could easily describe a system from exotic applications such as telecommand-telemetry functionality in a satellite down to everyday systems such as network interface management in parallel with a human-interactive computing element in a modern smartphone. These very different functionalities must coexist within the embedded system and are typically met by partitioning the software and functionality unto the heterogeneous platform. Typically timing sensitive functionality is assigned to a small microprocessor with a master role, which runs application software on a real time operating system including a task for communicating with the other processor. The data processing(in the satellite case) processor serves as a lower priority slave since it carries non critical functionality entering sleep states when possible suspending it's tasks.

1.6 Research questions

The core of this work amounts to an effort to answer the following questions. They are either explicitly or implicitly answered in the rest of this document, or wherever their complexity escapes the scope of this work, appropriate directions and thoughts on their resolution is proposed.

- **VM Portability** Can we execute application code, target independently in an embedded system?
- **Task migration** Is task migration across heterogeneous processing elements possible?
- **Task migration overhead** If so at what relative cost in performance?
- **Development effort** Can we reduce development and design effort with a VM solution?
- **Development effort – performance overhead tradeoff** Is the tradeoff balanced, i.e. do the benefits outweigh the incurred overhead?

2 Related work and Domain Analysis

In this section, an overview of the domain is presented. Additionally, to put this study into context, a synopsis of fundamental abstractions is needed, along with the relevant denominators that arise from them in the VM field. Namely, discussed are Application Portability and forward compatibility on Heterogeneous Systems, considerations on VM design and Interpreter based VMs.

2.1 VM based Application Portability and forward compatibility on Heterogeneous Systems

Virtual machines can provide a powerful abstraction over the underlying platform. From the point of view of the hardware, the interpreter itself is the only code executed. The VM hosted application code, the operands in local variables and on the operand stack and object representation on the heap are just inputs to the code executed natively on the hardware platform.

This mechanism allows applications to consider only the virtual platform the VM provides, while depending on the VM to support changes in the hardware. Especially in heterogeneous systems where the underlying hardware may be drastically updated in the future, it is very valuable to provide forward compatibility for the system's software. By deploying part or all of the application software on a VM, software can remain unchanged without redevelopment and porting when the hardware platform evolves.

Furthermore, the inherent abstraction in VMs can be expanded to provide application portability within the heterogeneous system across the deployed ISAs. This can allow with a small increase in complexity compared to solutions not based on a VM, for the composition of load balancing systems. Reconfiguring the software partitioning on the underlying ISAs, either statically or dynamically controlled by load balancing policies the system can effectively and with low development effort capitalize on the platform heterogeneity.

Towards employing a virtual machine to examine and enable these benefits for heterogeneous systems, we continue with a study on their components common architectures and associated tradeoffs.

2.2 Components and considerations on the design of VMs

A virtual machine (VM) is a high level abstraction on top of the native operating system, that emulates a physical machine. In this work, we are talking about process virtual machines and not system virtual machines. A virtual machine enables the same application to run on multiple operating systems and hardware architectures. The VMs for Java and lua can be taken as examples, where the code is compiled into their VM specific bytecode. The same can be seen in the Microsoft .Net architecture, where code is compiled into intermediate language for the CLR (Common Language Runtime).

What functions should a virtual machine generally implement? It should emulate the operations carried out by a physical CPU and thus should ideally fulfill the following concepts:

- Compile of source language into VM specific bytecode.
- Implement data structures to contain instructions and operands within memory.
- A call stack for function call operations.
- An virtual instruction pointer or program counter pointing to the next instruction to execute.
- A virtual cpu performing operations roughly equivalent to an emulator, the instruction dispatcher function roughly amounts to:
  - Fetching the next instruction pointed to by the instruction pointer.
  - Decode and fetches the operands.
  - Execute the instruction.

2.3 Virtual Machine Interpreters

Interpreters have a long history, being well studied and deployed in many runtime technologies. An interpreter can understand the source bytecode of a virtual ISA and interpret those virtual ISA instructions to those of a host platform. The interpreter-based VM abstracts away the underlying details of the host platform and makes the implemented high-level programming language portable across different hardware platforms as long as the VM has been ported to them. VM interpreters do not inherently have any dependency on specific features of the underlying platform or the operating system that hosts them. Thus they can be designed to be easily portable to several platforms and software stacks.

Listing 1: Switch interpreter dispatch

```
typedef enum {
add /* ... */
sub /* ... */
} Opcode;
void engine()
```

```
{
        static Inst program[] = { add /* ... */ };
        Inst *ip = program;
        int *sp;
        for (;;)
                switch (*ip++) {
                case add:
                        sp[1]=sp[0]+sp[1];
                        sp++;
                break;
                case sub:
                        sp[1]=sp[0]−sp[1];
                        sp++;
                break;
        /* ... */
        }
}
```

There are many different types of interpreters. Some interpreters simulate the ISA of new hardware, which does not yet exist, or to port binary applications compiled for one hardware platforms to run on another one. Some other interpreters (Java, , Lua, Perl, Tcl) are used to implement higher level programming languages. When an interpreter is used to implement a high-level language, there are two ways to convert the high-level source code into a sequence of virtual machine instructions or bytecodes understandable by the interpreter.

The translation of the source code into VM code can be either off-line as in the JVM, or during runtime as with Lua or Perl. VM instructions for higher-level portable languages like Java are usually designed with the intention of easing interpretation. The opcodes are usually encoded with one byte (256 possible VM instructions) in interpreters, such as Java and Smalltalk.

An interpreter is a very attractive option for VM implementation because it is easy to implement relatively to other execution engines and port to different platforms. However, they suffer from the drawback of low performance when compared to native code compiled directly from the source programming language. We will focus on the two categories of improvement which are relevant to this work. The first category is related to the interpreter implementation, such as the dispatch mechanism. The second category is related to the VM bytecode instruction architecture design choices, such as the choice between virtual a register machine instruction format or virtual stack machine one.

The core of a virtual machine (VM) is an execution engine, which behaves like a real processor. The execution engine, which can be implemented with an interpreter, fetches, decodes and executes VM instructions. Inside a virtual machine, an interpreter (the execution engine) has a virtual instruction pointer to the VM code currently being executed. In order to execute a VM instruction, the interpreter first fetches an instruction by using the instruction pointer, decodes the instruction (find the segment of code which implements the VM instruction in the interpreter loop), and then executes the code in the segment to carry out the function of the VM instruction. The last step includes the fetching the operands of the instruction and storing any results.

There are two types of operand locations for VM instructions. The first type of operand location is virtual registers or an operand stack, which are typically implemented as an array in the memory. The second type of operand location can be some data structures, such as an object representation, in the heap in the runtime data areas.

2.3.1 Register based vs Stack based VMs

Stacks are widely used in computer science. An evaluation stack is used to compute the value of arithmetic expressions. In a processor, a call stack saves the traces of subroutine calls and returns. A stack computer with a stack-based instruction set uses a stack to store the operands for instructions. In a stack computer, most of the instructions have implicit operands on the top of the operand stack. Any result produced by an instruction will be pushed onto the operand stack. There are two important instructions load and store. The load instruction pushes a value from an arbitrary RAM location onto the top of the computational stack and the store instruction saves a value from the top of the computational stack into a memory location.

A register machine uses the registers to store the operands (temporaries/result) of instructions. In a register machine, the operands must be encoded as part of an instruction. Most compilers for register architectures will use registers as much as possible because accesses to the registers are faster and a limited number of registers allow for shorter encoding of the instructions. Generating code for a register machine is more complex and a sophisticated register allocator is often needed to make the best use of a limited number of registers to maximize the performance of a source program.

There are essentially two main ways to implement a virtual machine: Based on a stack architecture, or register architecture. Prolific examples of stack based VMs are the Java Virtual Machine and the .Net CLR, as it is the widely used method for implementing virtual machines. Examples of register based virtual machines are the ones found in Lua, Perl(Parrot) and the Dalvik VM. The difference between the two approaches is in the mechanism used for storing and retrieving operands and their results.

Stack Based Virtual Machines A stack based virtual machine implements the general function of the VM with a memory structure where the operands are stored is a stack. Operations are carried out by popping data from the stack, processing them and pushing in back the results in a last in first out order. In a stack based virtual machine, the operation of adding two numbers would usually be carried out as follows:

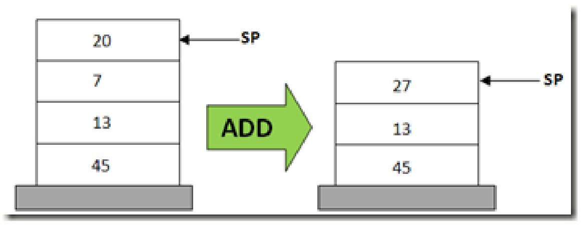


Figure 14: Simple stack machine

Because of the push and pop operations, four instructions are needed to carry out an addition operation causing significant overhead. An advantage of the stack based model is that the operands are addressed implicitly by the stack pointer(SP). This means that the Virtual machine does not need to know the operand addresses explicitly, as calling the stack pointer will pop the next operand. In stack based VMs, all arithmetic and logic operations are carried out by pushing and popping the operands and results in the stack.

Register Based Virtual Machines In the register based implementation of a virtual machine, the data structure where the operands are stored mirrors that of the registers of a cpu. There are no push or pop operations here, but the instructions need to contain the addresses (or symbol names) of the operands. In the register based architecture, the operands for the instructions are explicitly named in the instruction, unlike the stack based architecture, where we had a stack pointer pointing to the operand. For example, if an addition operation is to be carried out in a register based virtual machine, the instruction would look as follows:

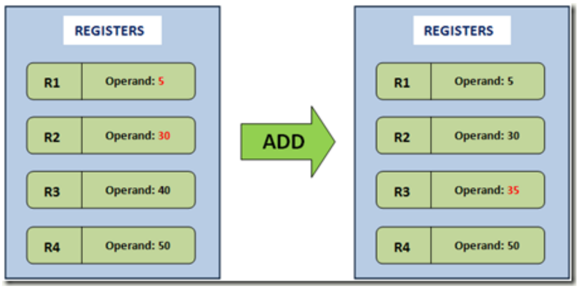


Figure 15: Simple register machine

As mentioned earlier, there is no pop or push operations and the related overhead, so the instruction for adding is simply one line. But unlike the stack, we need to explicitly mention the addresses of the operands as R1, R2, and R3. The advantage here is that the overhead of pushing to and popping from a stack is non-existent, and instructions in a register based VM execute faster within the instruction dispatch loop.

Another advantage of the register based model is that it allows for some optimizations that cannot be done in the stack based approach. One such instance is when there are common sub expressions in the code, the register model can calculate it once and store the result in a register for future use when the sub expression comes up again, which reduces the cost of recalculating the expression.

The problem with a register based model is that the average register instruction is larger than an average stack instruction, as we need to specify the operand addresses explicitly. Whereas the instructions for a stack machine are short due to the stack pointer, the respective register machine instructions need to contain operand locations, and results in larger bytecode size in register code compared to stack code.

Comparison of stack vs register VM implementation

There have been many arguments between stack and register-oriented instruction set architectures making it quite difficult to draw any definitive conclusions. below we list advantages and disadvantages, as well as some trends in mature implementations.

The main advantages of a stack-based instruction set are:

- Very high code density compared to other form of instruction sets resulting in small overall code size.
- Simplicity of the instruction set.
- Simple compiler implementation to generate stack-based code from the

source programming language.

In the case for register based Virtual Machines, there have been several that are mature and widely used, most notably Perl 6 Lua 5.0 and Dalvik, implemented with register instruction set architectures. Both Lua and Perl moved from a stack implementation to a register based VM implementation.

Parrot VM: The Parrot Virtual Machine for Perl 6 [18] moves away from its earlier stack-based versions to a new register architecture. It is intended to support multiple languages including Perl itself. It has many higher-level features such as objects, thread synchronization support and garbage collection.

The designers of Perl 6 give some of the following reasons for moving to the register architecture:

- Fewer register-based VM instructions are required than those of a stack VM.
- More research in optimization for register-based hardware to take advantage of.
- Break away from the tradition of stack VM architecture implementation to innovate.

Parrot originally used a scheme similar to a real processor. It had four groups (integers, floating-point numbers, strings and PMCs) of 32 registers. In the later evolution, the number of registers became unlimited to eliminate register spills. The virtual registers of the Parrot VM are stored in a register frame which could be pushed and popped onto a virtual register stack.

Lua 5.0: Lua[9] is a scripting language widely used in game industry. Lua 5.0[10] moved to register-based architecture partly because of earlier work on register machines in the developing group. There are 35 instructions in Lua's virtual machine. Virtual registers are kept in the run-time stack, which is implemented with an array. Constants and upvalues are also stored in arrays. Lua 5.0 uses 32 bits instruction encoding. The first 6-bits are the opcode. The next 8 bits are the first operand (A) and always present. The second (B) and third (C) operands are 9 bits. These second and third operands can be combined into one larger operand. Performance comparisons between Lua 5.0 (register based) and 4.0(stack based) show around a 20% improvement.

There are various ways to organize the virtual registers in a virtual machine. Some VMs such as earlier versions of the Parrot VM have a fixed number of general purpose registers or even a fixed number of registers for different data types, like a real processor. The state of the registers has to be saved and restored for function calls and returns. Another problem with a fixed number of registers is that a register allocator is needed and virtual register spilling can happen. This can cause a lot of unexpected memory copy operations. Other VMs create a new set of registers on a stack for each method call. Usually the number of required registers can be determined when compiling the source code. The number of addressable registers is limited by the size of operands (typically one byte). 256 registers are usually more than enough for modern object-oriented programming languages which encourage small methods. A register allocator is not needed, although one can be used to minimize the number of registers to save some space. Furthermore, all the VM registers are not physical registers in a real processor. They are typically represented using an array and indexed by an integer.

2.4 Techniques for efficient Interpretation

For an efficient interpreter for a general purpose language the design of choice is a virtual machine interpreter. The program is represented in an intermediate code that is similar in many respects to real machine code: the code consists of VM instructions that are laid down sequentially in memory and are easy to decode and process by software.

The interpretation of a virtual machine instruction consists of accessing arguments of the instruction, performing the function of the instruction, and dispatching (fetching, decoding and starting) the next instruction. The most efficient method for dispatching the next VM instruction is direct threading. Instructions are represented by the addresses of the routine that implements them, and instruction dispatch consists of fetching that address and branching to the routine. Direct threading cannot be implemented in ANSI C and other languages that do not have first-class labels, but GNU C provides the necessary features. Implementors who restrict themselves to ANSI C usually use the switch dispatch approach, VM instructions are represented by arbitrary integer tokens, and the switch uses the token to select the right routine.

When translated to machine language, direct threading typically needs three to four machine instructions to dispatch each VM instruction, whereas the switch method needs nine to ten. The additional instructions of the switch method over direct threading is caused by a range check, by a table lookup, and by the branch to the dispatch routine generated by most compilers.

2.4.1 Switch Dispatch

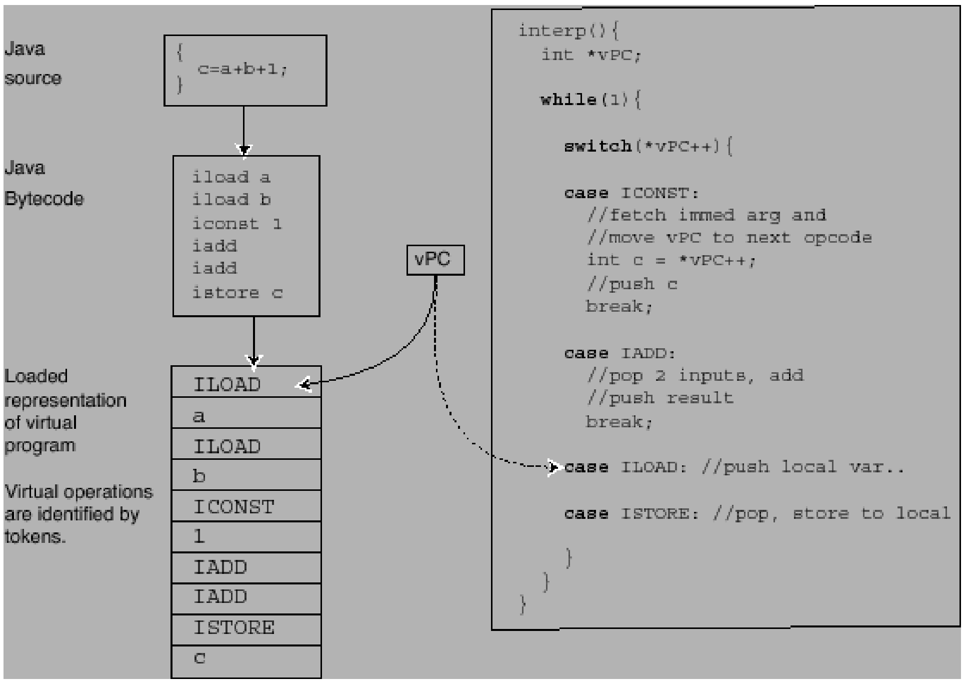
Switch dispatch, is the simplest and most portable dispatch mechanism. In this case we show as an example how an interpreter might choose a representation that is less compact than possible, for simplicity and speed of interpretation.

Listing 2: Switch Dispatch implementation generalized example

```
while (true) {
    byte instruction = instructionStream[programCounter];
    switch (instruction) {
        case NOP:
            programCounter += 1;
            break;
                /*
                 *
                *
                */
        case JUMP:
            programCounter = instructionStream[programCounter + 1];
            break;
    }
}
```

In the below figure, a loaded Java bytecode representation appears on the bottom left. Each virtual opcode is represented as a full word token even though a byte would suffice. Arguments, for those virtual instructions that take them, are also stored in full words following the opcode. This avoids any alignment issues on machines that penalize unaligned loads and stores.

Illustrated is the situation just before the statement c=a+b+1 is executed. The box on the right of the figure represents the C implementation of the interpreter. The vPC points to the word in the loaded representation corresponding to the first instance of iload. The interpreter works by executing one iteration of the dispatch loop for each virtual instruction it executes, switching on the token representing each virtual instruction. Each virtual instruction is implemented by a case in the switch statement. Virtual instruction bodies are simply the compiler-generated code for each case.


Java source
Java Bytecode
Loaded representation of virtual program
Virtual operations are identified by tokens.
vPC

```
{
  c=a+b+1;
}
```

```
iload a
iload b
iconst 1
iadd
iadd
istore c
```

```
ILOAD
a
ILOAD
b
ICONST
1
IADD
IADD
ISTORE
c
```

```
interp(){
  int *vPC;

  while(1){

    switch(*vPC++){

    case ICONST:
      //fetch immed arg and
      //move vPC to next opcode
      int c = *vPC++;
      //push c
      break;

    case IADD:
      //pop 2 inputs, add
      //push result
      break;

    case ILOAD: //push local var..

    case ISTORE: //pop, store to local

    }
  }
}
```

Figure 16: Switch interpreter – Java example

Above, a switch interpreter loading each virtual instruction as a virtual opcode, or token, corresponding to the case of the switch statement that implements it. Virtual instructions that take immediate operands, like iconst, must fetch them from the vPC and adjust the vPC past the operand. Virtual instructions which do not need operands, like iadd, do not need to adjust the vPC.

Every instance of a virtual instruction consumes at least one word in the internal representation, namely the word occupied by the virtual opcode. Virtual instructions that take operands are longer. This motivates the strategy used to maintain the vPC. The dispatch loop always bumps the vPC to account for the opcode and bodies that consume operands bump the vPC further, one word per operand. Although no virtual branch instructions are illustrated in the figure, they operate by assigning a new value to the vPC for taken branches.

A switch interpreter is relatively slow due to the overhead of the dispatch loop and the switch. Despite this, switch interpreters are commonly used in production (e.g. in the JavaScript and Python interpreters). Presumably this is because switch dispatch can be implemented in ANSI standard C and so it is very portable.

2.4.2 Direct Call Threading

Another portable way to organize an interpreter is to write each virtual instruction as a function and dispatch it via a function pointer. The following figure shows each virtual instruction body implemented as a C function. While the loaded representation used by the switch interpreter represents the opcode of each virtual instruction as a token, direct call threading represents each virtual opcode as the address of the function that implements it. So by treating the vPC as a function pointer, a direct call-threaded interpreter can execute each instruction in turn.

For historical reasons the name "direct" is given to interpreters which store the address of the virtual instruction bodies in the loaded representation. We assume that is because they can "directly" obtain the address of a body, rather than using a mapping table (or switch statement) to convert a virtual opcode to the address of the body. However, the name can be confusing as the actual machine instructions used for dispatch are indirect branches(in this case, an indirect call).

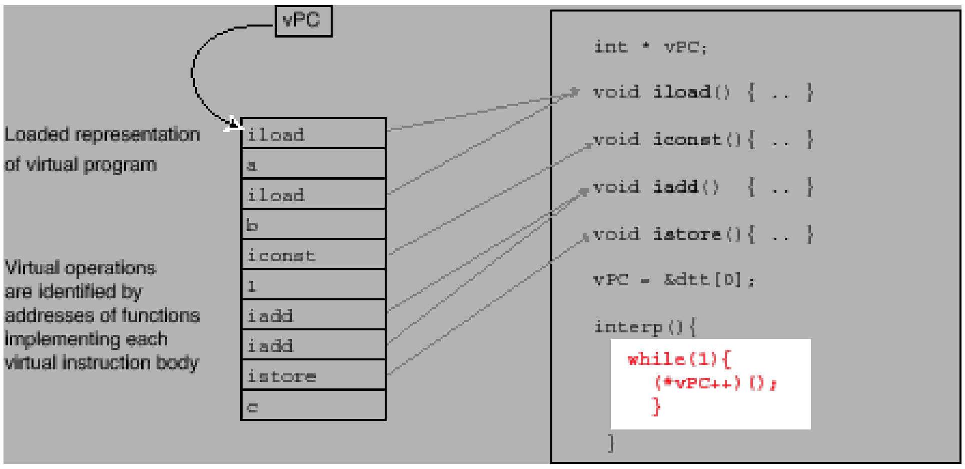


Figure 17: Call threading dispatch

In this figure, a direct call-threaded interpreter packages each virtual instruction body as a function. The shaded box highlights the dispatch loop showing how virtual instructions are dispatched through a function pointer. Direct call threading requires the loaded representation of the program to point to the address of the function implementing each virtual instruction.

Next we will describe direct threading, perhaps the most well-known high performance dispatch technique.

2.4.3 Direct Threading

Listing 3: Direct Dispatch generalized implementation example

```
op_ADD_INT:
    int op1 = READ_OP1;
    int op2 = READ_OP2;
    int result = op1 + op2; // The actual implementation.
    WRITE_RESULT(result);
    unsigned int opcode = pc->opcode;
    ++pc;
    goto *dispatch[opcode]; // Jump to code for next instruction.
```

In the above figure a generalized implementation of direct dispatch is shown. Note the goto statement jumping to the next opcode to execute and using it's label as a destination address.

In the below figure is a Direct-threaded Interpreter showing how Java Source code compiled to Java bytecode is loaded into the Direct Threading Table (DTT). The virtual instruction bodies are written in a single C function, each identified by a separate label. The double-ampersand && shown in the DTT is gcc syntax for the address of a label.

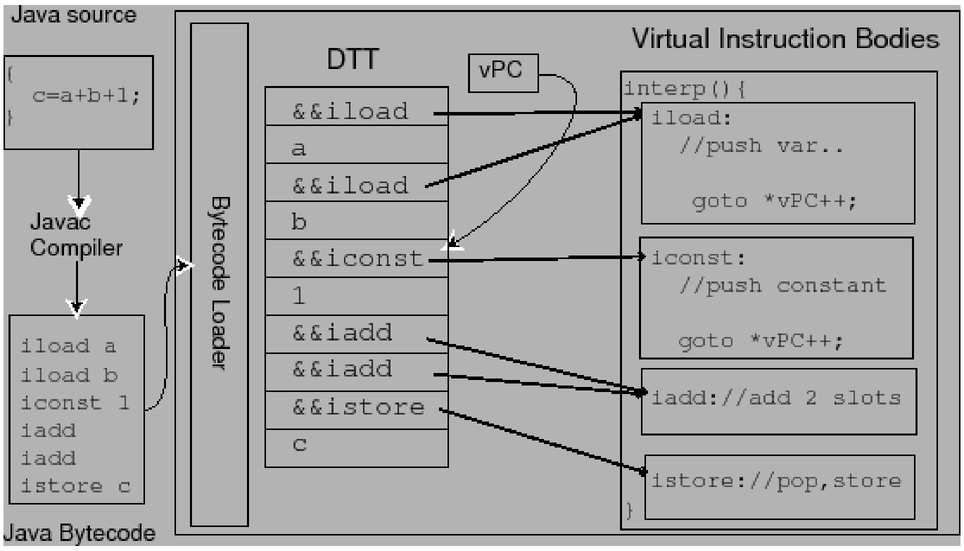
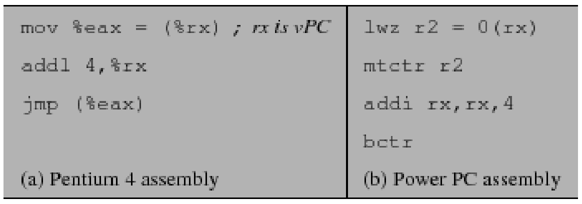


Figure 18: Direct threaded dispatch

Like in direct call threading, a virtual program is loaded into a direct threaded interpreter as a list of body addresses and operands. We will refer to the list as the Direct Threading Table, or DTT, and refer to locations in the DTT as slots.

Interpretation begins by initializing the vPC to the first slot in the DTT, and then jumping to the address stored there. A direct-threaded interpreter does not need a dispatch loop like direct call threading or switch dispatch. Instead, as can be seen in the figure, each body ends with goto *vPC++, which transfers control to the next instruction.

In C, bodies are identified by a label. Common C language extensions permit the address of this label to be taken, which is used when initializing the DTT. GNU gcc, as well as C compilers produced by Intel, support the label-as-value and computed goto extensions, making direct threading possible despite not being ANSI C.



mov %eax = (%rx) ; *rx is vPC*	lwz r2 = 0(rx)
addl 4,%rx	mtctr r2
jmp (%eax)	addi rx,rx,4
	bctr
(a) Pentium 4 assembly	(b) Power PC assembly

Figure 19: Machine instructions in Direct Dispatch

In the above figure machine instructions used for direct dispatch are shown. On both platforms assume that some general purpose register, rx, has been dedicated for the vPC. Note that on the PowerPC on the right, indirect branches are two part instructions that first load the ctr register and then branch to its contents.

Direct threading requires fewer instructions and is faster than direct call threading or switch dispatch. Assembler for the dispatch sequence is shown above. When executing the indirect branch the Pentium 4 will speculatively dispatch instructions using a predicted target address. The PowerPC uses a different strategy for indirect branches, as shown. First the target address is loaded into a register, and then a branch is executed to this register address. The PowerPC stalls until the target address is known, although other instructions may be scheduled between the load and the branch (like the addi in the previous figure) to reduce or eliminate these stalls.

2.4.4 The Dispatch technique impact on performance

In a detailed study on they effects of dispatch on performance in [16] it is found that generally interpreters perform an exceptionally high number of indirect branches. Typical C code performs significantly less than 1% non-return indirect branches; C++ programs (using virtual function calls) perform 0.5% – 2% indirect branches and other interpreters perform less than 1.5% non-return indirect branches. But up to 13% of the instructions executed in the dispatch mechanisms we examine are non-return indirect branches. Consequently, the performance of efficient virtual machine interpreters is highly dependent on the indirect branch prediction accuracy, and the branch misprediction penalty.

Without indirect branch prediction, the resulting mispredictions can take up most of the time even on a processor with a short pipeline. Even with indirect branch prediction, misprediction rates are remarkably high. In [16] profile guided static prediction only yields an average accuracy of 11%. Branch target buffers give accuracies of 2% to 50%, with a slight improvement for the two-bit variant. Two level predictors increase the performance of efficient

VM interpreters significantly, by increasing prediction accuracy to 82%98%. Threaded code interpreters are much more predictable than switch based ones, increasing accuracy from 2% – 20% to about 45%.

The reason is that a switched based interpreter has only a single indirect branch jumping to many targets, whereas a threaded code interpreter has many branches, each of them jumping to a much smaller number of frequent targets. Given that threaded code interpreters also require less overhead for instruction dispatch, they are clearly the better choice for efficiently implementing VM interpreters on modern processor architectures.

2.5 Dispatch mechanism tradeoff

As noted above the cost of all dispatch mechanisms is not the same. Threaded dispatch is about twice as fast as switch dispatch, although it cannot be implemented in ANSI C. Similarly, other interpreter optimizations which reduce the cost and/or number of dispatches will strongly affect the relative performance of stack and register architectures. So, specifically register machines might prove more efficient where the interpreter must be written in ANSI C for maximum portability, while a stack architecture might have an edge where GNU C or assembly language is acceptable. In effect the decision between threaded dispatch vs switch dispatch is that of portability vs performance.

2.6 Choosing a Bytecode and Intermediate Representation language

The most important design decision for the bytecode virtual ISA for a virtual machine is arguably that of stack based or register based. This is the most significant departure of virtual ISAs from actual real-processor ISAs. Considering the low level effects of this choice previously discussed at length, we now also consider this decision from a practical and high level standpoint.

Regarding Register based virtual instruction sets:

- Compilation to a register based ISA is significantly harder.
- Fewer register-based VM instructions are required than those of a stack VM.
- More research in optimization for register-based hardware to take advantage of.
- Very high code density compared to other form of instruction sets resulting in small overall code size.

- Simple compiler implementation to generate stack-based code from the source programming language.
- Simplicity of the instruction set

The main advantages of a stack-based instruction set are:

- Very high code density compared to other form of instruction sets resulting in small overall code size.
- Simplicity of the instruction set
- Simple compiler implementation to generate stack-based code from the source programming language.

3 The LLVM Framework

The Low Level Virtual Machine (LLVM) is an open source, mature optimizing compiler framework whose development started in 2000 seeing active development since. Today, it provides a high- performance static compiler backend, but can also be used to build virtual machines or just-in-time compilers and to provide midlevel analyses and optimization in a compiler pipeline. Its main innovation is in the area of life-long program analysis and optimization, it supports program analysis and optimization at compile time, link time, and runtime. At the time of writing, LLVM supports a wide array of target ISAs (backends). Although even more target backends are officially supported, the LLVM 3.4 trunk 202591 ships by default with the following:

- ARM
- Mips
- R600
- X86
- AArch64
- Hexagon
- NVPTX
- PowerPC
- Sparc
- SystemZ
- XCore
- CppBackend
- MSP430

The variety of the supported backends speak clearly to LLVM's versatility, with support from novel ISAs such as CUDA PTX to widely used microcontroller ISAs such as MSP430.

Over the last ten years, LLVM has substantially altered the compiler landscape . LLVM is now used as a common infrastructure to implement a broad variety of statically and runtime compiled languages (e.g., the family of languages supported by GCC, Java, .NET, Python, Ruby, Scheme, Haskell, D, as well as many lesser known languages). It has also replaced a broad variety of special purpose compilers, such as the runtime specialization engine in Apple's OpenGL stack. Finally LLVM has also been used to create a broad variety of new products, perhaps the best known of which is the OpenCL GPU programming language and runtime.

3.1 LLVM Architecture

The traditional three phase design – The most popular design for a traditional static compiler (like most C compilers) is the three phase design whose major components are the front end, the optimizer and the back end as in the following figure. The front end parses source code, checking it for errors, and builds a language-specific Abstract Syntax Tree (AST) to represent the input code. The AST is optionally converted to a new representation for optimization, and the optimizer and back end are run on the code.

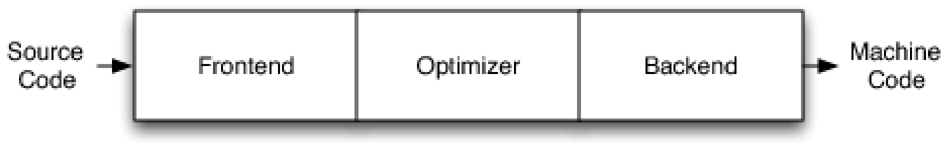


Figure 20: Simple Compiler

The optimizer is responsible for doing a broad variety of transformations to try to improve the code's running time, such as eliminating redundant computations, and is usually more or less independent of language and target. The backend (also known as the code generator) then maps the code onto the target instruction set. In addition to producing correct code, it is responsible for generating good code that takes advantage of unusual features of the supported architecture. Common parts of a compiler back end include instruction selection, register allocation, and instruction scheduling.

This model applies equally well to interpreters and JIT compilers. The Java Virtual Machine (JVM) is also an implementation of this model, which uses Java bytecode as the interface between the front end and optimizer.

Retargetability – The most important benefit of this classical design comes when a compiler decides to support multiple source languages or target architectures. If the compiler uses a common code representation in its optimizer, then a front end can be written for any language that can compile to it, and a back end can be written for any target that can compile from it, as shown below.

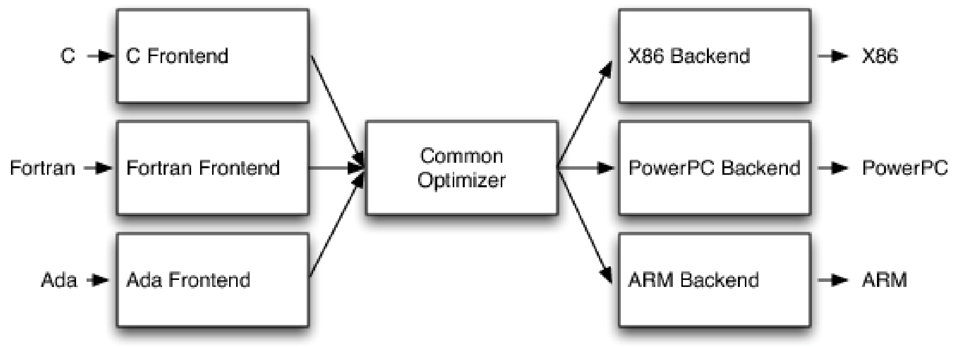


Figure 21: Retargetable Compiler

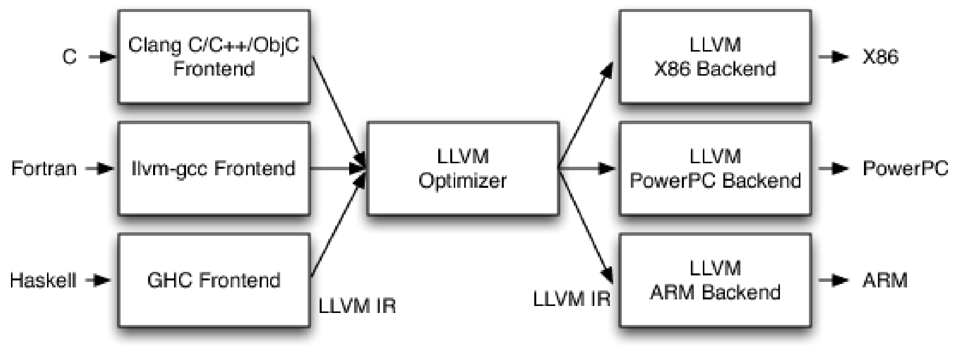


Figure 22: LLVM Compiler

With this design, porting the compiler to support a new source language requires implementing a new front end, but the existing optimizer and back end can be reused. If these parts weren't strictly separated, implementing a new source language would require starting over from scratch, so supporting N targets and M source languages would need N*M compilers.

Another advantage of the three-phase design which follows from retargetability is that the compiler serves a broader set of programmers than it would if it only supported one source language and one target. For a considerably large open source project, this means that there is a larger community of potential contributors to draw from, which naturally leads to more enhancements and improvements to the compiler. This is the reason why open source compilers that serve many communities (like GCC) tend to generate better optimized machine code than narrower compilers like FreePASCAL. This isn't the case for proprietary compilers, whose quality is directly related to the project's budget. For example, the Intel ICC Compiler is widely known for the quality of code it generates, even though it serves a narrow audience.

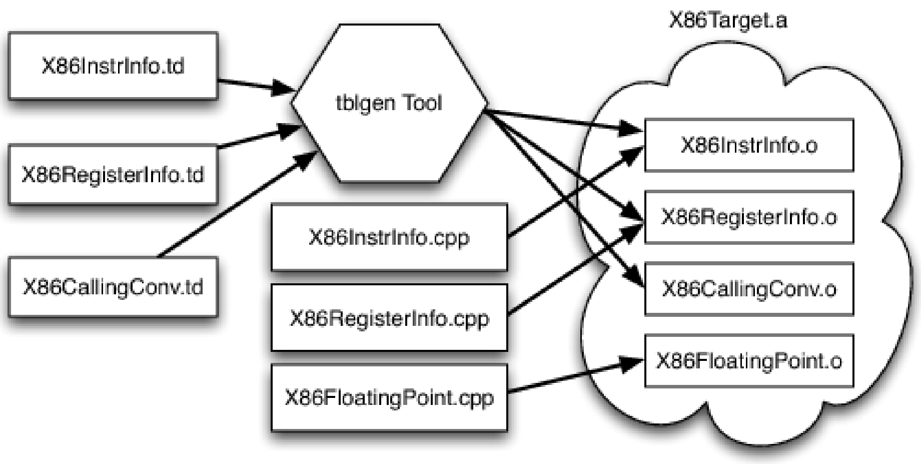


Figure 23: X86 Backend simplified

A final major benefit of the three phase design is that the skills required to implement a front end are different than those required for the optimizer and back end. Separating these makes it easier for front-end developers to enhance and maintain their part of the compiler. While this is a social issue, not a technical one, it matters a lot in practice, particularly for open source projects that want to reduce the barrier to contributing as much as possible.

While the benefits of a three-phase design are compelling and well-documented in compiler textbooks, in practice it is almost never fully applied. Looking across open source language implementations, one would find that the implementations of Perl, Python, Ruby and Java share no code. Further, projects like the Glasgow Haskell Compiler (GHC) and FreeBASIC are retargetable to multiple different CPUs, but their implementations are very specific to the one source language they support. There is also a broad variety of special purpose compiler technology deployed to implement JIT compilers for image processing, regular expressions, graphics card drivers, and other subdomains that require CPU intensive work.

That said, there are three major success stories for this model, the first of which are the Java and .NET virtual machines. These systems provide a JIT compiler, runtime support, and a very well defined bytecode format. This means that any language that can compile to the bytecode format, can take advantage of the effort put into the optimizer and JIT as well as the runtime. The tradeoff is that these implementations provide little flexibility in the choice of runtime: they both effectively force JIT compilation, garbage collection, and the use of a very particular object model. This leads to suboptimal performance when compiling languages that don't match this model closely, such as C.

A second success story is perhaps the most unfortunate, but also most popular way to reuse compiler technology: translating the input source to C code (or some other language) and send it through existing C compilers. This allows reuse of the optimizer and code generator, gives good flexibility, control over the runtime, and is really easy for front-end implementers to understand, implement, and maintain. Unfortunately, doing this prevents efficient implementation of exception handling, provides a poor debugging experience, slows down compilation, and can be problematic for languages that require guaranteed tail calls (or other features not supported by C).

A final successful implementation of this model is GCC4. GCC supports many front ends and back ends, and has an active and broad community of contributors. GCC has a long history of being a C compiler that supports multiple targets with support for a few other languages bolted onto it. As the years go by, GCC is slowly evolving to a cleaner design. As of GCC 4.4, it has a new representation for the optimizer (known as "GIMPLE Tuples") which is closer to being separate from the front-end representation than before. Also, its Fortran and Ada front ends use a clean AST.

Embedability and module reuse – While very successful, these three approaches have strong limitations to what they can be used for, because they are designed as monolithic applications. As one example, it is not realistically possible to embed GCC into other applications, to use GCC as a runtime/JIT compiler, or extract and reuse pieces of GCC without pulling in most of the compiler. People who have wanted to use GCC's C++ front end for documentation generation, code indexing, refactoring, and static analysis tools have had to use GCC as a monolithic application that emits interesting information as XML, or write plugins to inject foreign code into the GCC process.

There are multiple reasons why pieces of GCC cannot be reused as libraries, including rampant use of global variables, weakly enforced invariants, poorly-designed data structures, sprawling code base, and the use of macros that prevent the codebase from being compiled to support more than one front-end/target pair at a time. The hardest problems to fix, though, are the inherent architectural problems that stem from its early design and age. Specifically, GCC suffers from layering problems and leaky abstractions: the back end walks front-end ASTs to generate debug info, the front ends generate back-end data structures, and the entire compiler depends on global data structures set up by the command line interface.

In an LLVM-based compiler, a front end is responsible for parsing, validating and diagnosing errors in the input code, then translating the parsed code into LLVM IR (usually, but not always, by building an AST and then converting the AST to LLVM IR). This IR is optionally fed through a series of analysis and optimization passes which improve the code, then is sent into a code generator to produce native machine code, as shown in Figure 11.3. This is a very straightforward implementation of the three-phase design, but this simple description glosses over some of the power and flexibility that the LLVM architecture derives from LLVM IR.

3.1.1 LLVM as a collection of libraries

After the design of LLVM IR, the next most important aspect of LLVM is that it is designed as a set of libraries, rather than as a monolithic command line compiler like GCC or an opaque virtual machine like the JVM or .NET virtual machines. LLVM is an infrastructure, a collection of useful libraries of compiler technology that can be brought to bear on specific problems (like building a C compiler, or an optimizer in a special effects pipeline). While one of its most powerful features, it is also one of its least understood design points.

These libraries provide all sorts of analysis and transformation capabilities.They are expected to stand on their own, or explicitly declare their dependencies among other components if they depend on some other functionality to do their job.

Libraries and abstract capabilities are great, but they don't actually solve problems. The interesting part comes when someone wants to build a new tool that can benefit from compiler technology, perhaps a JIT compiler for an image processing language. The implementer of this JIT compiler has a set of constraints in mind: for example, perhaps the image processing language is highly sensitive to compile-time latency and has some idiomatic language properties that are important to optimize away for performance reasons.

The library-based design of the LLVM optimizer allows the implementer to pick and choose both the order in which transformation passes execute, and which ones make sense for the image processing domain in our example: if everything is defined as a single big function, it doesn't make sense to waste time on inlining. If there are few pointers, alias analysis and memory optimization are not a concern.

This is where the power of the library-based design of LLVM comes into play. It's straightforward design approach allows LLVM to provide a vast amount of capability, some of which may only be useful to specific audiences, without punishing clients of the libraries that just want to do simple things. In contrast, traditional compiler optimizers are built as a tightly interconnected mass of code, which is much more difficult to subset, reason about, and come up to speed on.

3.2 The LLVM Compiler IR

The LLVM assembly language, LLVM IR, is the input language which LLVM accepts for code generation. However, it also acts as LLVM's internal intermediate representation for program analysis and optimization passes. The IR has three equivalent representations: a textual representation (the assembly form), an in-memory representation, and a binary representation(bitcode). The textual representation is useful in a compiler pipeline where individual tools communicate via files, as well as for human inspection. The in-memory representation is used internally, but also whenever a compiler links to LLVM as a library to avoid the overhead of file input and output. The binary representation is used for compact storage – it occupies less storage than the textual format and can be read more efficiently.

The LLVM IR is low-level and assembly-like, but it maintains higher-level static information in the form of type and dataflow information – the latter due to using static single assignment (SSA) form. SSA form guarantees that every variable is only assigned once (and never updated), and hence, strongly related to functional programming[20]. The design goal in combining a low-level language with high-level static information is to retain sufficient static information to enable aggressive optimization, while still being low-level enough to efficiently support a wide variety of programming languages. The main features of LLVM's assembly language are:

- Low-level assembly with higher-level type information.
- Unlimited virtual registers, abstracting real hardware registers.
- Static single assignment form (SSA) with phi function.
- Functions and function calling with efficient tail call support.
- Explicit control flow with functions comprising blocks and branch statements.
- Direct memory access, as well as a type-safe address calculation instruction, getelementptr facilitating optimizations.

The single-assignment property of the SSA form requires the use of phi functions in the presence of low-level control flow with explicit branches. A phi function selects the value to be assigned to a virtual register in dependence on the edge of the control-flow graph along which execution reached the phi function. SSA form is well-established as a type of intermediate representation that simplifies the implementation of code analysis and optimization.

Listing 4: LLVM code to raise an integer to a power

```
define i32 @pow( i32 %M, i32 %N ) {
LoopHeader :
    br label %Loop
        Loop :
        %res = phi i32 [1, %LoopHeader], [%res2, %Loop]
        %i = phi i32 [0, %LoopHeader], [%i2, %Loop]
        %res2 = mul i32 %res , %M
        %i2 = add i32 %i, 1
        %cond = icmp ne i32 %i2 , %N
        br i1 %cond , label %Loop , label %Exit
        Exit :
        ret i32 %res2
}
```

The code in the above listing contains one complete LLVM function, which is made up of a list of basic blocks, each preceded by a label. The function has three basic blocks, those being LoopHeader, Loop, and Exit.

All control flow in LLVM is explicit, so each basic block must end with a branch (br) or return statement (ret). Variable names preceded by a percent symbol, such as %res and %i, denote virtual registers. Virtual registers are introduced by the unique assignment that defines them. All operations are annotated with type information, such as i32, which implies an integer type of 32 bits. Finally, the Loop block starts with two phi functions. The first one assigns to %res either the constant 1 or the value stored in register %res2 depending on whether execution entered the Loop block from the block LoopHeader or from Loop itself.

All LLVM code is defined as part of an LLVM module, with modules serving as compilation units. An LLVM module consists of four parts: meta information, external declarations, global variables, and function definitions. Meta information can be used to define the endianness of the module, as well as the alignment and size of various LLVM types for the architecture the code will be compiled to. Global variables are as expected, and are prefixed with the @ symbol, as are functions, to indicate that they are actually pointers to the data and have global scope. This also distinguishes them from local variables which are prefixed with the % symbol.

3.2.1 LLVM IR properties

Module structure – LLVM IR programs are composed of Modules, each of which is a translation unit of the input programs. Each module consists of functions, global variables, and symbol table entries. Modules may be combined together with the LLVM linker, which merges function (and global variable) definitions, resolves forward declarations, and merges symbol table entries. Following is an example of a "hello world" module:

Listing 5: An LLVM IR example Module

```
; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

; Definition of main function
define i32 @main() { ; i32()*
  ; Convert [13 x i8]* to i8 *...
  %cast210 = getelementptr [13 x i8]* @.str, i64 0, i64 0

  ; Call puts function to write out the string to stdout.
  call i32 @puts(i8* %cast210)
  ret i32 0
}

; Named metadata
!1 = metadata !{i32 42}
!foo = !{!1, null}
```

This example is made up of a global variable named .str, an external declaration of the puts function, a function definition for main and named metadata foo.

In general, a module is made up of a list of global values (where both functions and global variables are global values). Global values are represented by a pointer to a memory location (in this case, a pointer to an array of char, and a pointer to a function), and have one of the following linkage types.

Functions – LLVM function definitions consist of the "define" keyword, an optional linkage type, an optional visibility style, an optional DLL storage class, an optional calling convention, a return type, an optional parameter attribute for the return type, a function name, an argument list, optional function attributes, an optional section, an optional alignment, an optional garbage collector name, an optional prefix, an opening curly brace, a list of basic blocks, and a closing curly brace.

LLVM function declarations consist of the "declare" keyword, an optional linkage type, an optional visibility style, an optional DLL storage class, an optional calling convention, an optional unnamed_addr attribute, a return type, an optional parameter attribute for the return type, a function name, a possibly empty list of arguments, an optional alignment, an optional garbage collector name and an optional prefix.

A function definition contains a list of basic blocks, forming the CFG (Control Flow Graph) for the function. Each basic block may optionally start with a label (giving the basic block a symbol table entry), contains a list of instructions, and ends with a terminator instruction (such as a branch or function return). If an explicit label is not provided, a block is assigned an implicit numbered label, using the next value from the same counter as used for unnamed temporaries (see above). For example, if a function entry block does not have an explicit label, it will be assigned label "%0", then the first unnamed temporary in that block will be "%1", etc.

The first basic block in a function is special in two ways: it is immediately executed on entrance to the function, and it is not allowed to have predecessor basic blocks (i.e. there can not be any branches to the entry block of a function). Because the block can have no predecessors, it also cannot have any PHI nodes.

LLVM allows an explicit section to be specified for functions. If the target supports it, it will emit functions to the section specified.

An explicit alignment may be specified for a function. If not present, or if the alignment is set to zero, the alignment of the function is set by the target to whatever it feels convenient. If an explicit alignment is specified, the function is forced to have at least that much alignment. All alignments must be a power of 2.

If the unnamed_addr attribute is given, the address is known to not be significant and two identical functions can be merged.

Listing 6: LLVM IR function syntax

```
define [linkage] [visibility] [DLLStorageClass]
        [cconv] [ret attrs]
        <ResultType> @<FunctionName> ([argument list])
        [unnamed_addr] [fn Attrs] [section "name"] [align N]
        [gc] [prefix Constant] { ... }
```

Interesting instructions – Among the LLVM IR instructions the following have interesting properties and are either not typically found in other bitcode or IR languages, or have properties highly relevant to this work. For a complete reference of instructions, refer to the llvm documentation.

- **Terminators:** As mentioned previously, every basic block in a program ends with a terminator instruction, which indicates which block should be executed after the current block is finished. These terminator instructions typically yield a "void" value: they produce control flow, not values (withthe exception "invoke"). They are the following : "ret", "br", "switch", "indirectbr", "invoke", "resume" and "unreachable".

- **getelementptr:** The "getelementptr" instruction is used to get the address of a subelement of an aggregate data structure. It performs address calculation only and does not access memory.

  Listing 7: IR getelementptr syntax

  ```
  <result> = getelementptr <pty>* <ptrval>{, <ty> <idx>}*
  <result> = getelementptr inbounds <pty>* <ptrval>{, <ty> <idx>}*
  <result> = getelementptr <ptr vector> ptrval, <vector index type> idx
  ```

  The first argument is always a pointer or a vector of pointers, and forms the basis of the calculation. The remaining arguments are indices that indicate which of the elements of the aggregate object are indexed.

  The interpretation of each index is dependent on the type being indexed into. The first index always indexes the pointer value given as the first argument, the second index indexes a value of the type pointed to etc. The first type indexed into must be a pointer value, subsequent types can be arrays, vectors, and structs. Note that subsequent types being indexed into can never be pointers, since that would require loading the pointer before continuing calculation.

- **phi:** The "phi" instruction is used to implement the phi node in the SSA graph representing the function. The type of the incoming values is specified with the first type field. After this, the "phi" instruction takes a list of pairs as arguments, with one pair for each predecessor basic block of the current block. Only values of first class type may be used as the value arguments to the PHI node. Only labels may be used as the label arguments.

  There must be no non-phi instructions between the start of a basic block and the PHI instructions: i.e. PHI instructions must be first in a basic block.

  For the purposes of the SSA form, the use of each incoming value is deemed to occur on the edge from the corresponding predecessor block to the current block (but after any definition of an "invoke" instruction's return value on the same edge).

Listing 8: IR phi syntax

```
<result> = phi <ty> [ <val0>, <label0>], ...
```

3.3 On the suitability of compiler LLVM IR as a VM bytecode

It should be obvious from the above description that LLVM is a highly capable framework in support of building compiler toolchains and runtime environments. In this work we are interested specifically on the LLVM IR's use as a Virtual Machine input bitcode. Despite efforts in using it as such and the resulting Virtual Machine, it can be argued that it is not suitable for this task, namely that of building a platform. Building a platform meaning here that any system where LLVM IR is a format in which programs are stored or transmitted for subsequent use on multiple underlying architectures.

LLVM IR initially seems like it could well serve that purpose. It can appear highly attractive and it's documentation slightly misleading requiring an actual implementation effort to uncover these faults during development. There are several ways in which LLVM IR differs from actual platform ISAs, both high-level VM virtual bytecodes like Java or .NET and actual low-level ISAs like x86 or ARM.

First, the boundaries of what capabilities LLVM provides are vague. LLVM IR contains:

- Explicit target-specific ABI code. In order to interoperate with native C ABIs, LLVM requires front-ends to emit target-specific IR. This is sometimes avoidable, but not always.
- Explicitly target-specific features. As an example, x86_fp80's who's value in the IR's as a compiler IR is undeniable.
- Implicitly target-specific features. The most notable example being all the different Linkage kinds. These are all practically just gateways to features in real linkers, and native linkers vary quite a lot hence they are again required.
- Target-specific limitations in at first seemingly portable features. How big can the alignment be on an alloca? Or a GlobalVariable? What's the widest supported integer type? LLVM's various backends all have different answers to questions like these without a consistent implemen-

tation between them. This could possibly be avoided, but not without considerable collaboration among backend developers.

Even ignoring the fact that the quality of the backends in the LLVM source tree varies widely and specific IR feature implementation diverges, the question of "What can LLVM IR feature XY do?" has many backend-specific facets, sometimes unavoidably for it's use as a compiler IR.

Secondly, and perhaps more fundamentally, LLVM IR is a fundamentally vaguely specified language. It has:

- Undefined Behavior. LLVM is, at its heart, a C compiler, and Undefined Behavior is one of its cornerstones.

  High-level VMs like JVM or .NET typically raise predictable exceptions when they encounter program errors. Actual physical machines typically log their behavior extensively. LLVM IR is fundamentally different from both: it describes a bunch of rules to follow and then offers no description of what happens when they are violated. There are some analysis and correctness tools that can help locate violations of the rules in IR. But they can't find all undefined behaviors across code possibly generated by any front-end. There are even some kinds of undefined behavior that lack a method of detection for.
- Intentional vagueness. There is a strong preference for defining LLVM IR semantics intuitively rather than formally. This is very practical; formalizing the language is a considerable task and it reduces future flexibility. This would require even forbidding certain edge cases some transformations or backends rely on, requiring significant rewrites.

In stark opposition to Java's "write once, debug everywhere", consider the situation in LLVM IR, which is fundamentally opposed to even trying to provide that level of consistency. Furthermore, allowing optimizers to do target or subtarget specific optimizations, the chances of exposing edge cases and undefined behavior is increased.

Thirdly, LLVM is a low level system that doesn't represent high-level abstractions natively. It forces them to be partitioned up into many small low-level instructions.

- It makes LLVM's Interpreter really slow. The amount of work performed by each instruction is relatively small, so the interpreter has to execute a relatively large number of instructions to do simple tasks. Languages built for interpretation such as Lua do more with fewer instructions, and have lower per-instruction overhead.
- Similarly, it makes really-fast Just in time compiling very hard. LLVM

is fast compared to some static C compilers, but it's not fast compared to runtimes specifically built for JIT compilation. Compiling one LLVM IR instruction at a time can be relatively simple, but this approach generates very slow code. Fixing this requires recognizing patterns in groups of instructions, essentially complete optimization passes at runtime. This works, but it's more involved since it forces use of optimizations not designed for runtime.

In conclusion, consider the writing of an independent implementation of an LLVM IR Platform. The set of capabilities it should provide will end up depending on a combination of front-end and backend for which it is designed for. Semantic details would be vague, it would have to support features which require complicated infrastructure to implement which are rarely used. And to provide lightweight execution, it would need to translate the IR into something else better suited for it first. LLVM isn't actually a virtual machine. It's widely acknowledged that the name "LLVM" is a historical artifact which doesn't actually represent what LLVM actually grew to be. LLVM IR has actually evolved into a compiler IR.

4 Proposed Architecture and Design

To recap, the motivation for this work is to enable capitalizing on the benefits of heterogeneous embedded systems. We have elaborated on the utility of heterogeneous systems and the complexity they bring in real cost and development effort. We have considered the components and candidate implementation architectures of Virtual Machines and given a brief overview of the LLVM framework and it's capabilities.

In this section we draw the detailed requirements for a solution to the issues presented in the introduction. After a short presentation of the solutions and architectures initially considered, we present the proposed architecture: the Portable Heterogeneous llvm Ir Virtual Machine, PHIVM. The implementation of PHIVM is outlined along with the rationale behind related design decisions and a reconsideration of the stated requirements.

We can now formulate the requirements to a possible solution before discussing the proposed architecture and design pattern.

4.1 Requirements

Considering that the industry has not yet arrived to a consensus for managing the inherent complexity of heterogeneous systems, simply because only recent advances in integration and device programmability has made them prevalent. We require a solution to satisfy the following requirements to inform further research investigating such systems in the form of a design pattern or methodology, robust abstraction or reference implementation.

- **Architecture portability** The proposed solution should be easily ported to a wide variety of already existing or possibly useful future platforms. That is, the actual proposed software should be portable across ISAs and not be restricted to a certain software stack be it exotic libraries or OS dependencies.
- **Application portability** To enable evolution in the underlying platform the architecture should allow for applications to be easily, or ideally seamlessly adapted to changes in the hardware. This is sought to escape software development costs on porting while allowing for updates to the platform while it is deployed.
- **Task migration** It should allow for applications to move within the system's ISAs to satisfy different system behaviors. Crucially this mechanism should allow for runtime task migration at arbitrary application execution points. This should ideally be possible at several points in

the hosted application execution i.e. not restrictively at task or function level. The management of this functionality should allow to be performed by the user so that it can integrate with other parts of the system.

- **Flexibility** It should not restrict other design choices in the system. This means that it should not require for example specific communication, message passing frameworks, debugging/logging tools etc.
- **Embeddability** It should be easily embeddable without significant changes in the existing software stack and hardware platform. This implies a usable API embeddable in software as a simple library.
- **Development effort** It should overall reduce development effort in deploying and maintaining applications in heterogeneous systems.

4.1.1 Architecture portability

The most important requirement to support heterogeneity is portability of the proposed architecture itself. In the solution considerations subsection an initial effort that failed in this regard is described.

The proposed solution should be easily ported to a wide variety of already existing or possibly useful future platforms. That is, the actual proposed VM should be portable across ISAs and not be restricted to a certain software stack or hardware configuration. This is closely coupled with flexibility where the least amount of assumptions should be made in order to allow the user to fit the architecture to their own needs.

4.1.2 Flexibility and embeddability

It is required that the architecture does not restrict other design choices in the system. The possible configurations of the target heterogeneous systems are too many to predict and explicitly provide support for. It is understood that surely a task management functionality would exist along with some form of inter-process and/or inter-processor communication. The architecture should not assume a specific design pattern or framework for this and allow the user to adapt his own, via simple wrappers and essentially boilerplate code.

For this to be possible data structures to be communicated should be simple and well defined. Similarly the provided API should be lean to allow easy embeddability as well as being "hackable" to allow for adaptation to the user's needs. Although a vaguely defined term, "hackability" is well understood by most intended users of such a system. It implies the ability for the architecture to be easily understood and the codebase being manageable, allowing custom modifications by the end user to fit their needs.

In flexibility and avoiding restriction to other parts of the system we also include an almost free choice of input language. Perhaps the most important design decision, the programming language of the system is usually taken for granted, being restricted by frameworks or design patterns to be used in software. But in the general trend towards higher level languages, the architecture should support a wide array of candidate input languages. Obviously though, for practical reasons C/C++ support should be the main concern and exhibit complete robustness.

4.1.3 Application portability

To enable evolution in the underlying platform the architecture should allow for applications to be easily, or ideally effortlessly adapted to changes in the hardware. This to escape software development costs on porting while allowing for updates to the platform while it is deployed.

It is a fact that in general, software is never really finished and will have to adapt to future and unplanned behaviors. The same is true for complex hardware configurations, especially so when hosted on reprogrammable logic. To enable updates to the hardware, the architecture should allow for software to be ported to the new hardware configuration with minimal porting effort.

4.1.4 Exposing internal state for runtime application migration

As previously discussed, typically mixed constraints in system requirements have solutions residing on the opposite sides of tradeoffs. A popular tool to combat this is partitioning the application functionality on a heterogeneous platform. To empower this design pattern and adapt to different required behaviors of the system the architecture should provide a task migration capability.

This mechanism should allow applications to be moved around different processing elements of the system. Optimally at arbitrary points of the hosted application execution, this should be done seamlessly and without the hosted application's knowledge.

The management of this functionality should allow to be performed by the user so that it can integrate with other parts of the system. Furthermore the user should be able to provide their own rules for this mechanism to service their intended behavior profile. To enable this, hosted application runtime information should be exposed. In this way the user can build inspection tools to dynamically migrate the hosted application whenever it's behavior triggers it.

4.2 Solution considerations

In the beginning of this work it was clear that to meet the requirements posed, a Virtual Machine would be required. But the requirement for a flexible solution disallowed the use of mature VMs such as the JVM or highly embeddable and small VMs such as the lua VM. Such a choice would restrict the input language of the system to Java and lua respectively. Moreover, although the JVM is very mature and widely used, besides the input language problem it carries dependencies making porting to new platforms prohibitive failing on yet another requirement.

The other VM considered, the Lua VM, was investigated because of it's porting ease, being self contained with almost no dependencies on libraries and OS. The lua VM is also very easily embeddable in user code and a mature project overall. Eventually it too was rejected due to the input language (Lua) being too high level without alternative bytecode generators.

The requirement for flexibility in input languages, that is bytecode generating frontends, led to LLVM. As mentioned in the chapter describing it, LLVM frontends exist for practically all mainstream languages due to it's popularity, but there also exists a gcc backend targeting LLVM bytecode called dragonegg[1]. This practically makes gcc compiled languages also candidates for virtualization.

The LLVM framework provides an API for constructing specialized execution engines for LLVM bitcode(IR). This was used to construct an execution engine embedded in a virtual machine manager with the purpose of porting to embedded platforms. In the following listing we present as an example a part of the code of this effort to set up an LLVM Virtual Machine interpreter execution engine.

Listing 9: Example use of the LLVM framework for constructing an interpreter

```
LLVMContext &Context = getGlobalContext();
atexit(do_shutdown); // Call llvm_shutdown() on exit.
/*LLVM initialization
.
.
.
*/
// Load the bitcode...
SMDiagnostic Err;
Module *Mod = ParseIRFile(InputFile, Err, Context);
```

[1]Dragonegg(http://dragonegg.llvm.org) is a gcc backend that emits LLVM IR allowing for any language with a gcc frontend to produce LLVM bitcode

```
if (!Mod) {
    Err.print(argv[0], errs());
    return 1;
}
EngineBuilder builder(Mod);
builder.setMArch(MArch);
builder.setMCPU(MCPU);
builder.setMAttrs(MAttrs);
builder.setRelocationModel(RelocModel);
builder.setCodeModel(CMModel);
builder.setErrorStr(&ErrorMsg);
builder.setEngineKind(ForceInterpreter
        ? EngineKind::Interpreter
        : EngineKind::JIT);

// If we are supposed to override the target triple, do so now.
if (!TargetTriple.empty())
    Mod−>setTargetTriple(Triple::normalize(TargetTriple));

    CodeGenOpt::Level OLvl = CodeGenOpt::Default;
    switch (OptLevel) {
        default:
            errs() << argv[0] << ": invalid optimization level.\n";
            return 1;
        case ' ': break;
        case '0': OLvl = CodeGenOpt::None; break;
        case '1': OLvl = CodeGenOpt::Less; break;
        case '2': OLvl = CodeGenOpt::Default; break;
        case '3': OLvl = CodeGenOpt::Aggressive; break;
    }
builder.setOptLevel(OLvl);
TargetOptions Options;
Options.UseSoftFloat = GenerateSoftFloatCalls;
if (FloatABIForCalls != FloatABI::Default)
    Options.FloatABIType = FloatABIForCalls;
if (GenerateSoftFloatCalls)
    FloatABIForCalls = FloatABI::Soft;

    // Remote target execution doesn't handle EH or debug registration.
    if (!RemoteMCJIT) {
        Options.JITEmitDebugInfo = EmitJitDebugInfo;
        Options.JITEmitDebugInfoToDisk = EmitJitDebugInfoToDisk;
    }

builder.setTargetOptions(Options);
```

```
EE = builder.create();
if (!EE) {
    if (!ErrorMsg.empty())
        errs() << argv[0] << ": error creating EE: " << ErrorMsg << "\n";
    else
        errs() << argv[0] << ": unknown error creating EE!\n";
    exit(1);
}
// If the user specifically requested an argv[0] to pass into the program,
// do it now.
if (!FakeArgv0.empty()) {
    InputFile = FakeArgv0;
} else {
    // Otherwise, if there is a .bc suffix on the executable strip it off, it
    // might confuse the program.
    if (StringRef(InputFile).endswith(".bc"))
        InputFile.erase(InputFile.length() − 3);
}
// Add the module's name to the start of the vector of arguments to main().
InputArgv.insert(InputArgv.begin(), InputFile);

// Call the main function from M as if its signature were:
// int main (int argc, char **argv, const char **envp)
// using the contents of Args to determine argc & argv, and the contents of
// EnvVars to determine envp.
//
Function *EntryFn = Mod−>getFunction(EntryFunc);
if (!EntryFn) {
    errs() << '\'' << EntryFunc << "\' function not found in module.\n";
    return −1;
}
// If the program doesn't explicitly call exit, we will need the Exit
// function later on to make an explicit call, so get the function now.
Constant *Exit = Mod−>getOrInsertFunction("exit", Type::getVoidTy(Context),
        Type::getInt32Ty(Context),
        NULL);

// Reset errno to zero on entry to main.
errno = 0;
if (NoLazyCompilation) {
    for (Module::iterator I = Mod−>begin(), E = Mod−>end(); I != E; ++I) {
        Function *Fn = &*I;
        if (Fn != EntryFn && !Fn−>isDeclaration())
            EE−>getPointerToFunction(Fn);
    }
}
int Result;
// Trigger compilation separately so code regions that need to be
// invalidated will be known.
(void)EE->getPointerToFunction(EntryFn);
// Clear instruction cache before code will be executed.
if (RTDyldMM)
    static_cast<SectionMemoryManager*>(RTDyldMM)->invalidateInstructionCache();

    // Run main.
    Result = EE->runFunctionAsMain(EntryFn, InputArgv, envp);

    // Like static constructors, the remote target MCJIT support doesn't handle
    // this yet. It could. FIXME.

    // If the program didn't call exit explicitly, we should call it now.
    // This ensures that any atexit handlers get called correctly.
    if (Function *ExitF = dyn_cast<Function>(Exit)) {
        std::vector<GenericValue> Args;
        GenericValue ResultGV;
        ResultGV.IntVal = APInt(32, Result);
        Args.push_back(ResultGV);
        EE->runFunction(ExitF, Args);
        errs() << "ERROR: exit(" << Result << ") returned!\n";
        abort();
    } else {
        errs() << "ERROR: exit defined with wrong prototype!\n";
        abort();
    }
return Result;
```

Ultimately after considerable effort in building this virtual machine manager, this architecture was abandoned for the following reason: The most important requirement to support heterogeneity is portability of the virtual machine itself.

As a test of the development effort to do this with a VM using the LLVM API, it was attempted to port LLVM's libraries unto an ARM platform. This was of course a test because there are already available binaries for LLVM on ARM. Despite this ARM served as a "dry run" to assess the developer time needed for this. After considerable time and effort, porting was succesfull on ARMv6 with linux but not without considerable changes in the libraries' build system. As a result it was considered unreasonable to expect the resulting platform users to expend such porting effort for the multiple ISAs

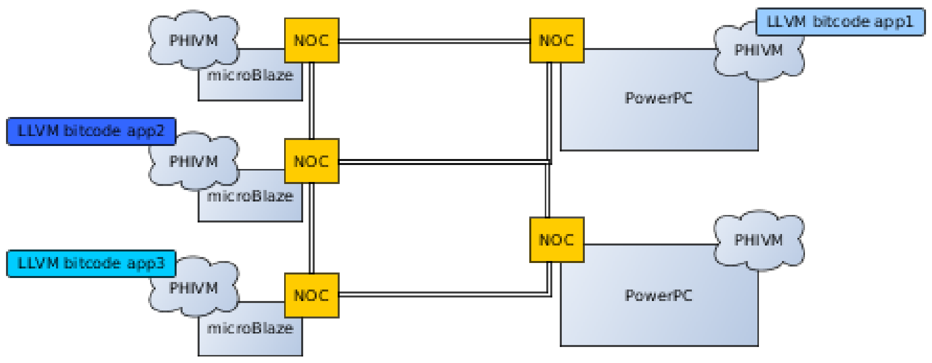


Figure 24: Instances of PHIVM in a Heterogeneous system

of their heterogeneous platform. This difficulty in porting of the LLVM framework is a result of its composing libraries being closely coupled. In practice one cannot only port e.g. the runtime parts (lli,jit) of the libraries only, without having to also port frontend related tools and libraries.

This failed attempt, served as a valuable lesson towards a final solution. This being that portability is not easily attained and it's difficulty scales exponentially with the size and complexity of software.

The next effort involved a simplified bytecode similar to [2]. This bytecode was inspired by Lua bytecode and was designed to be simple with the intention to make virtualization easier and low in memory usage. Development on an llvm backend to produce it was started, but not completed since the exploration process led to the realization for the current approach: why would we emit a custom bytecode from llvm IR and not interpret the IR directly?

4.3 The PHIVM approach

The proposed architecture to fulfill the stated requirements partially or fully is based on the Portable Heterogeneous llvm Ir Virtual Machine, PHIVM. PHIVM rests inside each processing element of the heterogeneous platform as a process on the operating system. It executes application code compiled to LLVM IR bitcode. It makes the least possible assumptions about the surrounding software and hardware platform to allow the platform designer to embed it within existing infrastructure. Furthermore, it exposes hosted application state to enable task migration guided by user designed tooling across the existing platform communication layer.

4.3.1 Allowing embedding

To be a viable design tool, PHIVM is structured as a library, intended to be embedded inside an existing code base. The library API exposes an initialization function and a main call to execute hosted bitcode. To use them the end user allocates in memory and passes to PHIVM the data structures, to be used during runtime, which are specified in the included headers. This is so that the user has complete control over the specifics of it's deployment and the way that an instance of PHIVM is hosted and managed natively.

4.3.2 Application migration and use of existing communication layers

To utilize the application migration capability, control of all the runtime context of hosted IR is given to the user. These data structures populated by the SSA IR registers(stack) memory(heap) and call stack of the hosted program are in the control of the user. The hosted program is using these virtualized versions of the stack, heap and call stack which allows for it to be interrupted and transferred to another instance of PHIVM without realizing it. The specifics of serialization and moving of the data structures are left to the user but have been designed to be very simple by the use of simple underlying data types, relatively flat hierarchy and lack of native pointers.

This mechanism is intended to avoid restriction by assuming a certain communication scheme. Even allowing serialization to plaintext and transfer by writing to a socket, this can accommodate most communication configurations between the processors comprising the platform.

Revisiting the previous figure of an abstract deployment of PHIVM, to put application migration in context consider the following:

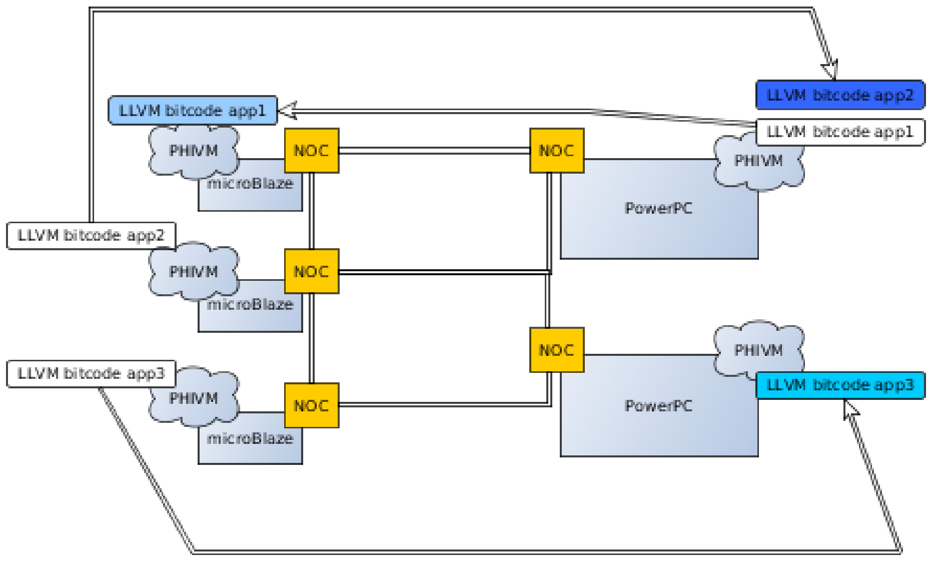


Figure 25: Application Migration over PHIVM instances

4.3.3 The case for a ground up VM design

The purpose of PHIVM is to eventually arrive at a reference implementation from which we can learn about how to meet the stated requirements. VM construction is not an undertaking to be taken lightly. Typically done by corporations employing large groups of specialized engineers, they reach maturity in a matter of years as in the case of the JVM(Sun) or Dalvik(Google) or by large communities of experienced researchers as with Lua or Perl.

To be able to expose internal application state and maintain VM and application portability meant no existing VM implementation could be extended or modified as discussed in 4.2. Since PHIVM serves the narrow purpose of a reference implementation, needing simplicity in it's design to allow end users to adapt it to their needs which would not be possible by extending an existing full-featured complex framework. In this vain PHIVM was designed from the ground up.

4.3.4 Effects of using the LLVM framework

To enable hosted applications to initially be in a wide array of input languages, PHIVM executes LLVM bitcode. The input high level code is statically compiled with an LLVM front-end and distributed in the form of bitcode for execution in PHIVM. As discussed in detail in chapter 3, there are many stable and mature LLVM front-ends and many more under development.

Through this, PHIVM is able to execute applications initially written in for example C/C++(clang/clang++) java(via the java frontend[2]) Python[3] and Haskell[4]. Most importantly, we are concerned with clang and clang++ since C/C++ are the most likely languages to be used. To testify to their maturity consider that FreeBSD, the OS focused on stability and security recently switched[5] it's default compiler to clang.

Our use of LLVM infrastructure frontends, brings another significant advantage: The use of the target independent optimization passes for LLVM bitcode. Since initial program code is passed from frontends constructed with LLVM, they can and are passed through a series of IR to IR optimization transformations ranging from simple dead code elimination up to constant propagation, alias analysis and more. This significant advantage was the main reason for the choice of LLVM as input bitcode, the wide range of usable and mature target independent optimization. More practically, it should be noted that because these are used across all frontends constructed with LLVM, they are very advanced, mature and constantly evolving.

Despite problems related to using LLVM IR as input bitcode to a virtual machine raised in subsection 3.3, the above benefits are too great to ignore and are a main contributor in the viability of PHIVM.

4.3.5 Component reuse from the LLVM framework

For the purpose of future proofing against future changes in the LLVM IR a component of the LLVM library was reused, the IR parser. Due to it's lack of documented strict semantics of instructions and formal grammar, constructing a complete parser of the IR from documentation alone is very difficult. It would require reverse engineering IR semantics from their frontend use, which was initially attempted but ultimately abandoned.

Regarding the portability of this code from LLVM and the effort required for compiling it to a new architecture, it required significant adaptation. The LLVM build system for this component was rewritten and it's implementation code refactored. In the case of major IR language changes in the future, a patch can be extracted and applied to the latest LLVM parsing component source without further changes to PHIVM. This way forward compatibility with LLVM IR is achieved along with easy porting to new architectures.

[2]https://llvm.org/svn/llvm-project/java/trunk/docs/java-frontend.txt
[3]http://pypy.org/
[4]http://www.haskell.org/ghc/docs/7.4.2/html/users_guide/code-generators.html
[5]`http://www.phoronix.com/scan.php?page=news_item&px=MTEwMjI`

4.4 Design decisions in the implementation of PHIVM

In this section we outline some notable design and implementation decisions taken in PHIVM. In general, simplicity was driving many design decisions for two reasons: As previously noted the construction of an execution engine is a big undertaking, so simpler solutions were favored to arrive to a reference implementation in reasonable time. For PHIVM to cover a broad range of uses, it is flexible through simplicity so that it can be easily modifiable by end users.

Instruction Dispatch – The heart of PHIVM, the instruction interpreter, is based on switch dispatch. Although the first version was using the direct dispatch mechanism, it was eventually abandoned in favor of the simpler and more portable switch dispatch. The labels as values mechanism direct dispatch is based on, is not ANSI standards compliant C and therefore not considered portable.

Listing 10: Part of the dispatch loop (trimmed)

```
std::vector<Instruction>::iterator Vpc;
    Instruction currInst;
    Vpc = BBtoexec->BBInstructionsV.begin();
    currInst = *Vpc;
    for (;;) {
        currInst = *Vpc;
        debugLog("opcode " << currInst.opcode);
        switch (currInst.opcode) {
        case OpCodeE::OP_ADD: {
            switch (currInst.type) {
            case Types::T_CHAR:
                getPtrSym(currInst, symContext, 0)->byteVal =\
                 getPtrSym(currInst, symContext, 1)->byteVal +\
                  getPtrSym(currInst, symContext, 2)->byteVal;
                debugLog(getPtrSym(currInst, symContext, 0)->byteVal);
                break;
            case Types::T_SHORT:
                getPtrSym(currInst, symContext, 0)->shortVal =\
                 getPtrSym(currInst, symContext, 1)->shortVal +\
                  getPtrSym(currInst, symContext, 2)->shortVal;
                debugLog(getPtrSym(currInst, symContext, 0)->shortVal);
                break;
            case Types::T_INT:
                getPtrSym(currInst, symContext, 0)->intVal =\
                 getPtrSym(currInst, symContext, 1)->intVal +\
                  getPtrSym(currInst, symContext, 2)->intVal;
                debugLog(getPtrSym(currInst, symContext, 0)->intVal);
                break;
                \*
                .
                .
                .
                */
        case OpCodeE::OP_NOP: {
            (++Vpc);
            break;
        }
        default: {
            debugLog(currInst.opcode << "\n");
            UnrecoverableErr("Invalid instruction", INSTR_ERR);
        }
        }
    }
    UnrecoverableErr("interpreter unreachable", BB_LOOP_ERR);
}
```

In this listing of some of the dispatching code in PHIVM a few things are visible:

- The instructions are matched in a switch, containing implementations for different types.
- After each execution of an instruction the virtual program counter is incremented.
- PHIVM having evolved from a direct dispatch interpreter it can easily be reverted back by removing the endless loop and replacing the switch statement with gotos. This can be done without further changes to the instruction bodies.
- There are debug macros included for most operations, which are enabled by a debug build.

In this kind of VM implementation practice, the layout of the switch and the code implementing the instruction operation results in eventually jumping to the compiler generated code for the instruction operation originally compiled for the code of the interpreter.

In memory representation of IR – The input IR is initially parsed with the parser component modified from LLVM and consequently stored in a data structure convenient for interpretation. As the following listings show, an enumeration is used to recognize the opcodes which are then stored in a vector of instructions where an instruction is a class in the second listing. Basic blocks along with their labels as well as functions, are identified in a similar way, with std::vectors containing their comprising instructions and basic blocks respectively. It should be noted that basic blocks are also recognized by the terminator instructions in their end which are treated appropriately to resolve a possible upcoming phi instruction in the basic block that follows it.

Listing 11: A snippet of the opcodes class

```
enum class OpCodeE {

    OP_ADD,
    //<result> = add <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = add nuw <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = add nsw <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = add nuw nsw <ty> <op1>, <op2> ; yields {ty}:result
    OP_FADD,
    //<result> = fadd [fast-math flags]* <ty> <op1>, <op2> ; yields {ty}:result
    OP_SUB,
    //<result> = sub <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = sub nuw <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = sub nsw <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = sub nuw nsw <ty> <op1>, <op2> ; yields {ty}:result
    OP_FSUB,
    //<result> = fsub [fast-math flags]* <ty> <op1>, <op2> ; yields {ty}:result
    OP_MUL,
    //<result> = mul <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = mul nuw <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = mul nsw <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = mul nuw nsw <ty> <op1>, <op2> ; yields {ty}:result
    OP_FMUL,
    //<result> = fmul [fast-math flags]* <ty> <op1>, <op2> ; yields {ty}:result
    OP_UDIV,
    //<result> = udiv <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = udiv exact <ty> <op1>, <op2> ; yields {ty}:result
    OP_SDIV,
    //<result> = sdiv <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = sdiv exact <ty> <op1>, <op2> ; yields {ty}:result
    OP_FDIV,
    //<result> = fdiv [fast-math flags]* <ty> <op1>, <op2> ; yields {ty}:result
    OP_UREM,
    //<result> = urem <ty> <op1>, <op2> ; yields {ty}:result
    OP_SREM,
    //<result> = srem <ty> <op1>, <op2> ; yields {ty}:result
    OP_FREM,
    //<result> = frem [fast-math flags]* <ty> <op1>, <op2> ; yields {ty}:result

    OP_SHL,
    //<result> = shl <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = shl nuw <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = shl nsw <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = shl nuw nsw <ty> <op1>, <op2> ; yields {ty}:result
    OP_LSHR,
    //<result> = lshr <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = lshr exact <ty> <op1>, <op2> ; yields {ty}:result
    OP_ASHR,
    //<result> = ashr <ty> <op1>, <op2> ; yields {ty}:result
    //<result> = ashr exact <ty> <op1>, <op2> ; yields {ty}:result
    OP_AND,
    //<result> = and <ty> <op1>, <op2> ; yields {ty}:result
    OP_OR,
    //<result> = or <ty> <op1>, <op2> ; yields {ty}:result
    OP_XOR,
    //<result> = xor <ty> <op1>, <op2> ; yields {ty}:result
    ...
    ...
```

Listing 12: The instruction class

```
struct Instruction {
    OpCodeE opcode;
    cmp_cond cond;
    Types type;
    std::vector<Op> opsV;
};
```

Tagged unions – All values including registers and memory are held in tagged unions. Although indirectly a tagged union comprises usually of a struct containing a value inside a union such as the one in the following listing as well as a tag, noting its type for accessing the union correctly. In PHIVM an equivalent method is used albeit without an explicit struct. Values are held in a union and type information is held with a type tag but they are implicitly matched, via the index in the instruction operands vector. This is such as to allow the same functionality to service other IR objects such as labels and instruction options.

Listing 13: PHIVM Value union (trimmed)

```
union Value {
    byte_ty byteVal;
    short_ty shortVal;
    int_ty intVal;
    float_ty floatVal;
    double_ty doubleVal;
    void_ptr_ty ptrTy;
    Types typeVal;
    std::vector<Value> vectorVal;
\*
 * ...
 */
    func *ptrFunc;
    double VmemPtr;
    BB *ptrBB;
    //std::string symbolID;
};
```

Listing 14: PHIVM type tags (trimmed)

```
enum class Types {
    T_CHAR,
    T_SHORT,
    T_INT,
    T_FLOAT,
    T_DOUBLE,
\*
 * ...
 */
    T_VECTOR,
    T_BBPTR,
    T_LABELPTR,
    T_VOID,
    T_PTR
};
```

SSA registers and virtual memory – Registers are stored in an stl vector of Values with their type and scope information held by the instruction that uses them. Similarly the virtual memory stores type information and values in an stl vector which is controlled by memory instructions in the IR. The use of tagged unions and stl vectors is repeated in other aspects of PHIVM for the following reasons:

- Stl vectors are mostly portable and recognized by type aware message passing tools.

- As a data structure it is the most easily serializable and target data layout independent.
- The stdc++ native implementation is responsible for the details of memory management.

Listing 15: Operand struct scope enum

```
enum scopeE {
    S_GL,
    S_FN,
    S_BB
};

struct Op {
    scopeE scope;
    int indx;
    int length;
};
```

In a final note, the above type-value-index pattern is repeatedly used across PHIVM. It is used to avoid pointers in the data structures which are meant to be transferred for task migration. This pattern replaces the use of native pointers which are of course not portable, albeit with a certain overhead.

5 Design evaluation

In this section we consider the initial requirements stated in the previous chapter and how PHIVM evaluates against them. We discuss the challenges in evaluating PHIVM, it being a framework rather than an implementation on a specific platform. Continuing we present a comparison with similar tools that could perform some of the functionalities of PHIVM in certain configurations. This section closes with a short note on how and whether PHIVM can prove valuable in further exploration of this domain.

5.1 Reconsidering the initial requirements against PHIVM

In the previous section, a set or requirements were posed, for a solution to the complexities of heterogeneous systems. These were in short:

- **Architecture and application Portability**
- **Flexibility and embeddability**
- **Task migration**
- **Development effort**

Concerning Architecture and application portability:
PHIVM succeeds in this regard, architecture portability being a design concern from the start in it's implementation. It carries no dependencies to other software that cannot be easily resolved with an already provided mechanism i.e. the parsing component reused from LLVM.

Application portability being a mostly inherent property of Virtual Machines, is also satisfied. In the typical abstraction a virtual machine provides, hides hardware implementation details from hosted software, allowing already noted benefits such as hardware platform evolution, independent of deployed software.

Flexibility and embeddability of the implementation:
Being a framework, that is software purposed to be used to build other software, it satisfies embeddability, providing a clean API and exposing it's inner workings only when necessary to make wrapping it in existing software easy.

Flexibility though is only partially satisfied. Although PHIVM makes only generalized assumptions to enable cooperation with other software it is restricted in it's assumption that it will run inside a linux environment. This is though only an artificial restriction, since there is no practical reason for not supporting any real time operating systems or baremetal deployment. This restriction emerges from it's development and testing being done on linux. Besides this, it does not restrict restrict the communication layer or the system or the input language. Unusually for VMs it allows practically any mainstream programming language to provide executable bitcode by using LLVM as a frontend.

Task migration:
Task migration is also possible with PHIVM, by it being designed to expose hosted application state upstream to the software that embeds PHIVM. Despite this we consider this requirement only partially satisfied. The requirement of flexibility somewhat limits PHIVM to the extent where it can support migration. This is due to it not requiring a specific serialization or communication interface present in the system, it cannot provide a complete migration solution, leaving parts of it to be fulfilled by the user of PHIVM with the communication framework or memory topology they have available.

Development effort:
In this requirement it fully succeeds by allowing platform developers to separate hardware maintenance and updates and software maintenance and updates. This is again a benefit of it being a Virtual Machine. Application development can rest on the virtualized abstraction PHIVM provides, without maintaining complex cross-compilation toolchains for every part of the heterogeneous system.

It is worthy to note that there exists a tradeoff between, task Migration and development effort against flexibility of the implementation. In both cases the later was chosen. We preferred to attempt increased flexibility rather than restricting platform properties to propose a more complete task migration solution. As for development effort it would be expended anyway for deploying the platform, so there was no comparable benefit to make platforms on which PHIVM is viable, more narrow.

5.2 Considerations on testing

PHIVM is a reference implementation of a framework making it quite difficult to formally benchmark. There are a few reasons for this:

- Being a framework it is purposed to enable building a certain class of software. This means that a complete application would have to be built utilizing a subset of the provided capabilities. Then this application would have to be formally benchmarked against something else.
- But it's capabilities are largely novel in the domain of embedded systems and not usable or required in general purpose computing i.e. homogeneity is the standard and heterogeneity has a completely different meaning (software using both CPU and GPGPU).

There is no other readily available framework or similar application for PHIVM to compare against. Even if a platform was built using PHIVM it is unclear which parameters would be useful to actually measure. The reason being that a platform built with PHIVM is closely coupled with the other parts comprising it. This makes it difficult to separate the contributions to performance of PHIVM or the other technologies that platform is built with.

Consider as an example, trying to measure the application migration overhead using PHIVM, in a hardware platform of two different ISAs connected with message passing DMA buffers managed by OpenMPI. There are at least four contributors to this namely the size of the hosted application context, the latency of the buffering and locking mechanism in hardware, the latency of OpenMPI and lastly the overhead of PHIVM. It is not clear how these could be separated and normalized to produce any meaningful conclusion.

Even if the above problem, of isolating PHIVM's impact on system behavior, were somehow solved there is still the issue of having nothing to compare against. That would again require another implementation of the same functionality but implemented somehow differently without PHIVM. There is no readily available framework for this, which would result in a highly specialized, custom and complex system further obscuring any meaningful results.

Such a complex and customized implementation simply for the purpose of comparison, escapes the scope of this thesis both in time and complexity. In place of formal benchmarking and comparison for the above reasons, what follows is a feature comparison of PHIVM and other software that could possibly be modified to serve a similar purpose.

5.3 Feature and functionality comparison

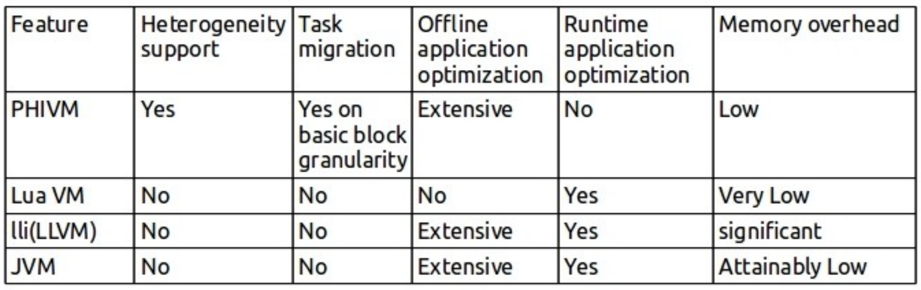
In this section we consider the features and functionality provided by PHIVM through a comparison. For the reasons outlined above, formalized benchmarking is not applicable due to the novelty of PHIVM's capabilities and the difficulty in extracting meaningful conclusions from such testing.

It is arguably more valuable to consider PHIVM's properties through its compliance to the requirements that emerged from our discussion of a viable platform. Despite this, to make the PHIVM approach more clear we put it in the context of existing tools and frameworks that could possibly serve a similar purpose in some hardware platform configurations. The tools considered are virtual machines or frameworks that could be used to build them. They could alternatively be used with some modifications or extensions in similar scenarios.

On including Lua it should be noted that although not being a framework for building VMs, it is included as a representation of a family of similarly virtualized languages(Perl, Python etc.). Its design is also considered viable for deployment in embedded systems and as such it was considered initially in chapter 4.

Specifically in the context of managing the complexity of heterogeneous systems we consider:

- PHIVM
- The Lua VM
- lli the LLVM infrastructure interpreter
- The default JVM



Feature	Heterogeneity support	Task migration	Offline application optimization	Runtime application optimization	Memory overhead
PHIVM	Yes	Yes on basic block granularity	Extensive	No	Low
Lua VM	No	No	No	Yes	Very Low
lli(LLVM)	No	No	Extensive	Yes	significant
JVM	No	No	Extensive	Yes	Attainably Low

Figure 26: Feature comparison table

To put these properties better into context for the VM suitability for deployment in an embedded system:

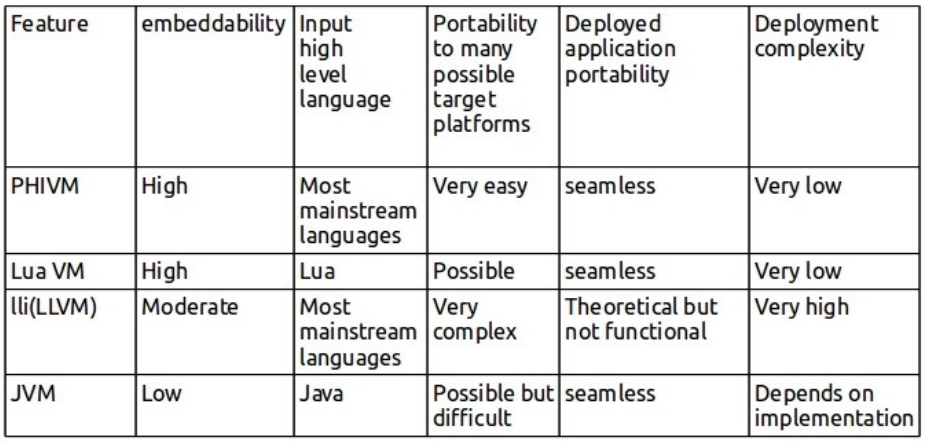
- **Heterogeneity Support –** PHIVM has been developed with this in mind, basing all design decisions around it. The other frameworks considered could possibly be modified for deployment in a heterogeneous platform but not without considerable modifications. Such modifications being an afterthought, would very likely severely limit their other features, making them practically unusable.

- **Task migration –** Task migration is a novel capability, especially on heterogeneous systems. PHIVM provides this by exposing hosted application state at basic block granularity, allowing migration at basic block boundaries. It is possible that lli could be modified to allow equivalent functionality, at the function level, but not without severely limiting or disabling other features. Namely lli's runtime optimizations would have to be disabled since they work across function boundaries (which is otherwise a benefit).

- **Offline application optimization –** By exploiting available LLVM frontends PHIVM is equal to lli in this regard, having possibly the same input bitcode. There is a wide array of target independent LLVM IR transformations they can employ for optimization. On the other hand Lua being compiled to bytecode at runtime cannot support this. Finally the JVM being mature is also comparable with LLVM in static compile time optimizations and hence considered equal.

- **Runtime optimization and Memory overhead –** Runtime optimization and memory overhead are closely coupled since runtime optimization expends considerable memory to transform bytecode in parallel to execution. Lli and and the JVM both expend memory for runtime optimization as well as other features not relevant to embedded systems. JVM is more mature in this compared to LLVM simply because of its prevalence, having sparked alternate implementations and configurations to improve this. In this regard PHIVM and the Lua VM provide low memory overhead mostly because they are based on simple execution engines on compact bytecode. Lua fares better though in this tradeoff, providing the option for a trace Just in Time compiler.

5.3.1 System designer effort



Feature	embeddability	Input high level language	Portability to many possible target platforms	Deployed application portability	Deployment complexity
PHIVM	High	Most mainstream languages	Very easy	seamless	Very low
Lua VM	High	Lua	Possible	seamless	Very low
lli(LLVM)	Moderate	Most mainstream languages	Very complex	Theoretical but not functional	Very high
JVM	Low	Java	Possible but difficult	seamless	Depends on implementation

Figure 27: Comparison of developer effort – table

Here we consider framework properties that relate to developer effort, NRE and maintenance cost in using these in real world platforms:

- **Embedability –** Embeddability inside other software is relevant to allow the system designer to consider this as an extension to their platform. Most importantly it implies that such a framework can be added after design decisions on other parts of the system and surrounding software already built. PHIVM and Lua evaluate positively in this mostly due to their simplicity and their being designed for use as embeddable libraries. LLVM, although it is too a library it presents too much complexity in deploying it hence requiring effort to cooperate with other software. The default JVM is not designed for this and is intended mainly as standalone software.
- **Input programming language –** No restriction for input programming language in PHIVM and lli exists due to their use of frontends built with the rest of the LLVM framework. As discussed in previous chapters, LLVM has emerged as an excellent framework for compiler backends, the "social" result being frontend designers use it to capitalize on this. The Lua VM is restricted to Lua while the JVM is designed specifically for Java. It is actually possible to target Java bytecode to compile a few languages(Clojure, Scala), but these are close Java

relatives. Using java bytecode drives decisions in their higher level features and makes java bytecode frontends difficult to develop for other languages, due to its being specifically designed for Java(the language).

- **Portability to target platforms and deployment dependencies** – These two properties are closely tied since dependencies on other software and functionality is the main reason for non-portability. PHIVM and Lua owning to their simplicity are mostly independent of other functionality offered by foreign software. Hence they are flexible and portable due to being largely self contained. LLVM and the JVM in antithesis, have complex dependencies on libraries and OS services.
- **Deployment complexity** – In overall deployment complexity PHIVM is the simplest having been designed for this. Lua is also simple to deploy and use although not in heterogeneous systems which is not its purpose. Again we mention LLVM's complexity which is due to its large feature set, only a small part of which is usable in the platforms of interest in this study. Finally JVM depends on the specific implementation on this, having many different choices in different levels of maturity and design purpose.

5.4 Evaluation as a research and exploration enabler

In completion of this section it is worthy to note the value of PHIVM as a research enabler. Heterogeneous systems have only recently become prevalent due to advances in integration and device programmability. They have not been widely deployed to their fullest and as a result the industry has not yet arrived in a consensus for managing their inherent complexity. This work is experimental in the hopes that the lessons learned and the tradeoffs uncovered here, will prove valuable to further exploration.

6 Conclusion

In conclusion, this work has fulfilled its purpose to provide a reference implementation and present the design tradeoffs in doing so. With the domain analysis initially performed, the potential family of solutions converged clearly towards a Virtual Machine after stating the requirements.

Keeping in mind the characteristics of heterogeneous embedded systems, the requirements for a virtual machine were focused into what became PHIVM. The tradeoffs of designing a virtual machine for our purposes emerge through the proposed architecture and design decisions that comprise PHIVM.

The reference implementation of PHIVM exhibits the feasibility of managing the complexity of heterogeneous systems and benefiting from new design methodologies in doing so, namely task migration. Perhaps most importantly by providing this capability, PHIVM enables more research towards providing for dynamic behaviors and adaptability by migration.

Although this implementation served well for these purposes, it is not based in production quality runtime frameworks, hence it is only intended as a reference implementation. Its purpose is to realize the approach presented in this study and inform further research investigating heterogeneous systems in the form of a design pattern or methodology and appropriate abstractions.

Finally, this text could prove to be a resource to any future work on the matter, having outlined the technologies relevant to this domain, presenting the challenges faced throughout the work and also discussing the rationale of the design choices.

References

[1] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[2] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke, "LLVA: A Low-level Virtual Instruction Set Architecture," in *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*, (San Diego, California), Dec 2003.

[3] R. Herveille, "Wishbone system-on-chip (soc) interconnection architecture for portable ip cores," tech. rep., OpenCores Organization, 2010.

[4] D. U. Becker, *Efficient microarchitecture for network-on-chip routers.* PhD thesis, Stanford University, 2012.

[5] S. Nanda and T. cker Chiueh, "A survey of virtualization technologies," tech. rep., 2005.

[6] A. Donovan, R. Muth, B. Chen, and D. Sehr, "Pnacl: Portable native client executables," tech. rep., Google, Feb 2010.

[7] R. L. Bocchino Jr and V. S. Adve, "Vector llva: a virtual vector instruction set for media processing," in *Proceedings of the 2nd international conference on Virtual execution environments*, pp. 46–56, ACM, 2006.

[8] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, (Palo Alto, California), Mar 2004.

[9] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho, "Lua-an extensible extension language," *Softw., Pract. Exper.*, vol. 26, no. 6, pp. 635–652, 1996.

[10] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "The implementation of lua 5.0," vol. 11, pp. 1159–1176, jul 2005.

[11] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Security and Privacy, 2009 30th IEEE Symposium on*, pp. 79–93, IEEE, 2009.

[12] D. Ehringer, "The dalvik virtual machine architecture," *Techn. report (March 2010)*, 2010.

[13] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, "A survey of adaptive optimization in virtual machines," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 449–466, 2005.

[14] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 4, p. 2, 2008.

[15] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron, "The case for virtual register machines," in *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pp. 41–49, ACM, 2003.

[16] M. A. Ertl and D. Gregg, "The behavior of efficient virtual machine interpreters on modern architectures," in *Euro-Par 2001 Parallel Processing*, pp. 403–413, Springer, 2001.

[17] M. A. Ertl and D. Gregg, "Optimizing indirect branch prediction accuracy in virtual machine interpreters," in *ACM SIGPLAN Notices*, vol. 38, pp. 278–288, ACM, 2003.

[18] F. Fagerholm, "Perl 6 and the parrot virtual machine," 2005.

[19] S. J. Lee, D. K. Raila, and V. V. Kindratenko, "Llvm-chimps: Compilation environment for fpgas using llvm compiler infrastructure and chimps computational model," *Proceedings of 4th Annual Reconfigurable Systems Summer Institute. Urbana, USA*, pp. 1–10, 2008.

[20] A. W. Appel, "Ssa is functional programming," *SIGPLAN notices*, vol. 33, no. 4, pp. 17–20, 1998.

[21] H. Muhammad and R. Ierusalimschy, "C apis in extension and extensible languages.," *J. UCS*, vol. 13, no. 6, pp. 839–853, 2007.

[22] G. Wilson, "The architecture of open source applications."