A real-time virtual machine implementation for small microcontrollers

Article *in* Innovations in Systems and Software Engineering · September 2012

DOI: 10.1007/s11334-012-0188-1

3 authors:

Roger Davis
MITRE
1 PUBLICATION **2** CITATIONS
SEE PROFILE

Phillip A. Laplante
Pennsylvania State University
275 PUBLICATIONS **2,552** CITATIONS
SEE PROFILE

Bo Ingvar Sanden
Colorado Technical University
65 PUBLICATIONS **264** CITATIONS
SEE PROFILE

A Real-Time Virtual Machine Implementation For Small Microcontrollers

W. Roger Davis, Colorado Technical University

Phillip A. Laplante, Penn State

March 2012

Abstract

A new software artifact to host a full general purpose virtual machine interpreter on a very small microcontroller platform is described. This machine is shown capable of performing a full suite of general capabilities as well as enhanced capabilities efficiently by abstracting the VM instruction set. Although there was an element of subjectivity in determining what defines improved capabilities (performance can be measured), the study was largely quantitative in nature. Measurements were made on the execution of software programs in the virtual machine while running on the target platform in order to demonstrate the machine's capabilities. Additionally, multitasking capabilities were added to the baseline and found to perform efficiently within the VM. The results proved to be satisfactory and demonstrate that a robust virtual machine can be made available for very small embedded platforms based on simple microcontrollers.

Keywords: virtual machine, real-time systems, microcontrollers

1 Historical Background

A virtual machine (VM) is a software program that emulates a computer processor, whether that processor actually exists in hardware form or not. Modern virtual machines can be divided into two classes, system and application level virtual machines. The former is closely tied to the underlying hardware and attempts to emulate it exactly, often with hardware support. The latter is a software interpreter that executes bytecodes (virtual instructions) to emulate a real or imaginary processor.

Virtual machine technology was first developed for IBM mainframe computers in the 1960s to manage multiple virtual platforms (Creasy, 1981). Early virtual machines ran primarily at the system (hardware) level and managed the entire machine to make it appear that there were many instances of the hardware. In the early 1970s Niklaus Wirth developed the Pascal language which compiled into portable interpreted P-Code and thus created the first application virtual machine (Wirth, The Programming Language Pascal, 1970). A major advantage of the P-Code VM was its portability. A Pascal application could be built and tested on one kind of machine and then the P-Code could be run on another machine as long as it hosted a P-Code interpreter. This early experiment gave rise to the modern byte code interpreters that exist today on various platforms. With the advent of the personal computer in the 1980s, the concept of running virtual code was limited to BASIC interpreters and occasionally a P-Code or Modula M-Code interpreter (Wirth, Programming in Modula-2, 1983). This limitation was because the personal computer platform lacked the power and the hardware support to host a more sophisticated VM. As personal computers became more powerful application virtual machines became more common with Sun's Java Virtual Machine (JVM) in the late 1990s followed by Microsoft's Common Language Runtime (CLR) in 2001 (Microsoft, 2010). These environments allow programs to be written and compiled for Java (in the case of JVM) or C# (in the case of CLR) on one platform and then run on a another platform supporting the corresponding VM. Inferno is another virtual environment that was created by Bell Labs for distributed systems. It uses the *Dis* virtual machine and the Limbo language (Dis Virtual Machine Specification, 1999).

More recently, the concept of a system VM also saw a resurgence as the personal computer became more powerful and hardware support was added to support virtualization.

System virtual machines, and hypervisors, such as VMWare and Xen could transparently emulate the entire machine and support multiple operating systems.

2 Virtual Machines on Embedded Platforms

Application virtual machines have migrated to high end embedded platforms, which now have powerful processors and megabytes of memory. For example, many smart phones such as the iPhone, Droid, and Blackberry now run implementations of JVM, LLVM (Low Level Virtual Machine), or Dalvik as the virtual environment to support user applications. Applications are written in Java or other languages and compiled into bytecodes for the VM target. These bytecode programs are then downloaded by users to their phones and executed there. Other former desktop virtual machines have found their way onto embedded platforms as well. Microsoft recently released an open source version of their .NET Micro Framework that requires about 200K of code space (Microsoft, 2011). This allows third party embedded hardware platforms to run the CLI virtual machine interpreter and to develop applications using managed code such as C#. Similarly, an open source version of .NET called Mono was developed by Novell and is maintained by Xamarin. This has been ported to the Android platform to run on smart phones and tablets (Mono for Android, 2012).

Many additional VMs have come into existence in recent years that support personal computers and high end embedded platforms. For example, Parrot was created for Perl and is also used for Ruby and Python as a scripting language platform that primarily runs on personal computers (Parrot, 2010). Lua was developed at PUC-Rio in Brazil (Lua the Programming Language, 2010). It is a small dynamic scripting language platform that has been adopted by the game industry as middleware for game engines and also has an emdedded version called eLua (eLua - Embedded Lua, 2010).

Virtual machines have also found limited usefulness on much smaller microcontroller platforms. A microcontroller is a type of microprocessor that integrates various hardware components onto a single integrated chip. Microcontrollers vary greatly in size and performance and the smaller ones generally have a Harvard architecture as shown in Table 1-1. At the lowest end of the spectrum there are devices with code and data space measured in hundreds of bytes and often have a performance rating below 1 DMIPS (Dhrystone Million Instructions per Second). These devices are used in very special devices that typically perform one task using a simple state machine. These are hereby designated as "tiny" microcontrollers. Smartcard processors also generally fall into this category. Smartcards can be low power devices such as the STMicroelectronics ST23YL80 which has a large code capacity in ROM but very little RAM (STMicroelectronics, 2008). Smartcards can also be very high powered devices. Although the SC300 ARM Cortex M3 that is also found in smartcards has a powerful processor and a large code space, it can address only 30K of RAM and is thus listed in the tiny category (STMicroelectronics, 2011).

At the other end of the embedded microcontroller spectrum are products like the Texas Instruments Stellaris version of the ARM Cortex-M3 (Texas Instruments, 2010) which is rated at about 100 DMIPS and can address large amounts of memory. As such, it might be considered a "large" microcontroller. These high end microcontrollers are commonly found in medical equipment and industrial control systems..

Company / Family	Model	Internal Code	Internal RAM	Max Addr	Register Size	Max Clock / DMIPS	Arch	Category
Freescale (Motorola)								
6805	68HC05X32	32K	528	N/A	8 bit	22 MHz	VN	Tiny
6808	68HC908GP32	32K	512	64K	8 bit	8 MHz	VN	Tiny
6811	68HC11Ex	20K	768	64K	8 bit	3 MHz	VN	Tiny
6816	68HC16Y1	48K	2K	1M	16 bit	16.78 MHz	VN	Large

ST Microelectronics								
ST23	ST23YL80	396K	6K	N/A	16 bit	10 MHz	H	Tiny
ARM Cortex-M3	SC300	1.28M	30K	N/A	32 bit	22.5 MHz	H	Tiny
Microchip								
PIC10	PIC10F200	256x12	16	N/A	8 bit	4 MHz	H	Tiny
PIC18	PIC18F1320	8K	256	N/A	8 bit	8 MHz	H	Tiny
	PIC18F87K90	128K	4K	N/A	8 bit	64 MHz	H	Tiny
PIC24	PIC24FJ256	256K	96K	16M	16 bit	32 MHz / 16 DMIPS	H	Small
PIC32	PIC32MX	512K	128K	4G	32 bit	80 MHz /120 DMIPS	H	Large
Atmel								
ATtiny	ATtiny10	1K	32	N/A	8 bit	12 MHz	H	Tiny
	ATtiny167	16K	512	N/A	8 bit	16 MHz	H	Tiny
ATmega	ATxMega128A1	128K	8K	16M	8 bit	32 MHz /6 DMIPS	H	Small
	ATxMega256A1	256K	32K	16M	8 bit	32 MHz /6 DMIPS	H	Small
AT32	AT32UCA	512K	64K	128 M	32 bit	66 MHz /91 DMIPS	H	Large
Texas Instruments								
ARM Cortex-M3	LM3S9B96	256K	96K	64M	32 bit	80 MHz /100 DMIPS	H	Large

Table 1-1 Sample Spectrum of Microcontroller Technology

In the middle is a selection of "small" microcontrollers. These are commonly 8 or 16 bit devices with about 128K to 256K of flash memory. They usually have a fair amount of internal RAM such as 64K or 128K, and they often can address a large amount of RAM externally. Their performance is generally rated between 1 and 20 DMIPS. Most microcontrollers in the tiny and small range also have modified-Harvard architectures. As microcontrollers, they are often equipped with many built-in peripherals for performing various functions. Many of them, such as the Atmel AVR xMega family have several multi-purpose serial ports, timers, analog to digital converters, encryption engines, and sophisticated interrupt mechanisms (Atmel Corporation, 2010). These are usually used in small communication systems, machine controllers, automotive applications, and robotics. Attempts at hosting a VM on the tiny or small category of microcontroller have resulted in implementations that have limited functionality. As shown in Figure 1-1, most modern application virtual machines do not target tiny or small microcontroller platforms. The VM code footprint is too large, often by magnitudes.

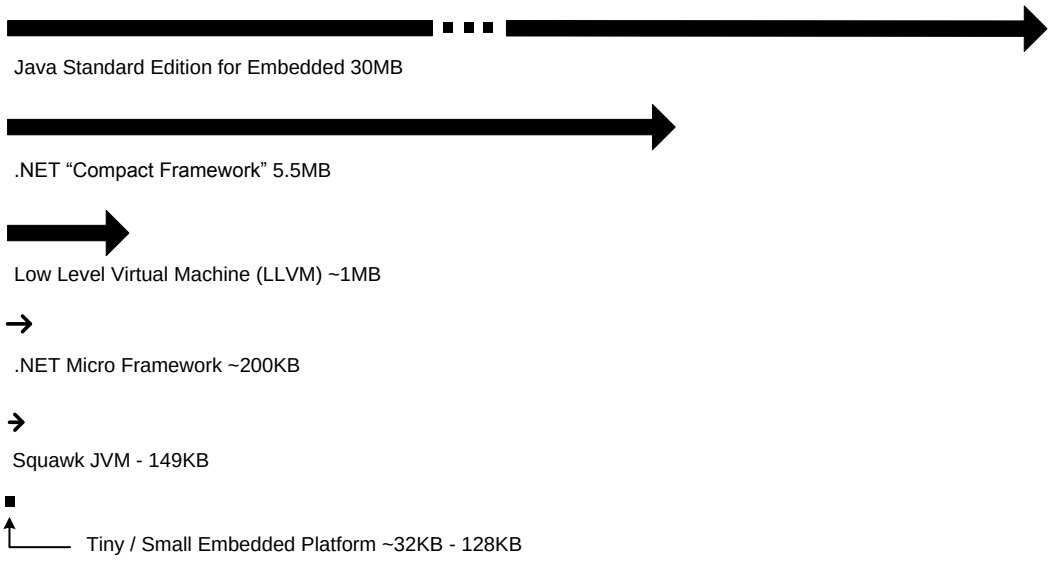

Figure 1-1 Size Comparison of Popular Virtual Machines and a Small Platform

Additionally the smaller microcontroller platforms lack the power and resources to run the VM fast enough for practical use. Some VMs improve speed through just-in-time (JIT) compilation which fundamentally will not work on the Harvard architectures found in many microcontrollers that cannot execute native code from RAM. Just-in-time compilation converts the bytecodes into native instructions on the fly that can then be executed by the processor at native speed to improve performance dramatically. Harvard architectures, by definition, execute code from code space which is typically flash memory, not data space. JIT compilation will result in having native code compiled into data space unless of course that native code is then flashed into code memory before execution, which would defeat much of the purpose of JIT compilation.

2.1 Examples of Limited Virtual Machines on Small Microcontrollers

There have been some examples where limited virtual machines have been made to execute on tiny and small microcontrollers. In each case significant compromises were made to

the VM so that it would fit in the limited space available. Researchers at Berkeley developed the Mate' VM that runs on tiny sensor node platforms with only 8K to 128K of instruction memory and 512 bytes to 4K of RAM (Levis & Culler, 2002). PICOBIT is a VM that runs the dynamic language Scheme on a Microchip PIC18 microcontroller platform (St-Amour & Feeley, 2009) (Abelson, et al., 1998).

There have also been many implementations of Java on small platforms. In each case, features were removed from the Java VM and many times post-processing of the executable was required to remove unnecessary functionality resulting in proprietary Java implementations. One of these is the Darjeeling virtual machine (Brouwers, Langendoen, & Corke, Darjeeling, a feature-rich VM for the resource poor, 2009). Similarly, TakaTuka post processes the Java instructions to remove unused parts of the interpreter and to optimize the application for each specific application, making it incompatible with standard Java (Aslam, Schindelhauer, Ernst, Spyra, Meyer, & Zalloom, 2008). Another implementation which is also a small subset of Java is NanoVM (Harbaum, 2006) (Praus & Kastner, 2008). This implementation is a greatly reduced Java subset to fit on an AVR8 which has only 8K of flash code memory. The primary purpose is to provide the capability to control a robot which is controlled by the microcontroller and has support for I/O, string processing, and low level control. ParticleVM is another Java subset for a small PIC processor (Riedel, Arnold, & Decker, 2007). The AmbiComp virtual machine is an implementation that requires post-processing (Eickhold, Fuhrmann, Saballus, Schlender, & Suchy, 2008). Similarly, Jelatine is a J2ME-CDLC Java implementation that requires post-processing (Agosta, Reghizzi, & Svelto, 2006). The Squawk JVM was developed to overcome the size and resource problems of embedded targets (Simon, Cifuentes, Cleal, Daniels, & White, 2006). Java classes are post-processed into "suites" which compacts the bytecodes and pre-links

the classes for position independent execution from read-only memory. The footprint of the interpreter without the extra library features was reduced to 149K bytes.

2.2 Limitations of Virtual Machines for Small Microcontrollers

As has been demonstrated in the literature, various attempts to host application virtual machines on small microcontroller platforms have been met with limited success. Java implementations generally require significant pre and post processing. Additionally, dynamic linking, garbage collection and loss of portability limit the usefulness for small platforms. Downloading a new optimized interpreter to match each application to a target, for example, defeats many of the advantages of using a virtual machine.

Specialized virtual machines such as Mate' serve only their narrow intended purpose by providing a very limited instruction set suited for just one purpose and do not provide a general purpose computing environment.

Dynamic scripting languages such as Lua (Lua the Programming Language, 2010) are not suited for deeply embedded microcontroller applications because they use precious CPU cycles to perform operations that a static language performs at compile time. They provide a usefulness in interactive sessions but not in deployed static applications where microcontrollers are commonly used. As an example, a static language can interpret a communications data stream using a simple structure overlay to extract the desired data, whereas languages such as Lua need special runtime operations to deserialize the data packet. This sort of processing is best done at compile time when the resources are limited.

3 A Simplified General Purpose Virtual Machine for Small Embedded Platforms

Java and other languages that have been rehosted on large embedded platforms are not always a good fit for smaller platforms. As a result, many of the more powerful features are often stripped away to make the application and the interpreter fit onto the target. This is a common theme in the research where special post-processing is performed to attempt to reduce the footprint and to speed up the initial execution. At the other end of the spectrum, virtual machines with extremely limited usefulness are often employed to meet a specific need such as Mate' for sensor networks.

Commonly, small microcontrollers require a general purpose virtual machine that provides a fully functional instruction set but it must operate in an extremely small footprint. The need for certain features may or may not be needed such as file I/O, floating point operations or a network stack depending on the environment, but the needs of small microcontrollers are generally different than those of their larger counterparts.

Similarly, it has been shown in the research that it is desirable to start with an existing virtual machine and then extend it to add domain specific capabilities rather than to start from scratch (Hudak, Building domain-specific embedded languages, 1996).

4 The Research

4.1 Research Strategy

The research that was conducted involved developing a new software artifact to host a full general purpose virtual machine (VM) onto a small microcontroller platform. The goal was to determine whether a virtual machine can run on the platform and whether it can provide enhanced capabilities and improved performance. Although there was an element of subjectivity in determining what defines improved capabilities (performance can be measured), the study was

largely quantitative in nature. Measurements were be made on the execution of software programs in the virtual machine while running on the target platform.

4.2 Research Chronology

The research was divided into five phases so that the schedule and accomplishments could be properly managed.

4.2.1 Phase One – Select Candidate Platform and Virtual Machine

The first phase was to select a representative small embedded microcontroller platform and a VM to use in the research. The speed and size of the microcontroller platform was selected to fit within the limitations defined in this study for a *small* target. The selected target occupied the low end of the definition since if the hypotheses could be proven to be true for the low end of the range, it is reasonable to extrapolate that they will be true at the higher end of the *small* range as well.

In selecting a VM for this study, it can be argued that the optimal path would have been to design a custom VM to maximize the likelihood of finding a suitable match. With this strategy, the VM would be designed from the ground up to run within the constraints of the target platform. It would be a daunting task to design such a system within the timeframe of this research. There are many open source VMs that are presently available and one need not find the optimal VM for the research as long as a representative one can be found that can be modified sufficiently to test the hypotheses. Thus, an effort within the first phase is to define a set of criteria that a VM needs to provide in terms of size, speed, and features and then find the candidate that most closely meets those criteria.

4.2.2 Phase Two – Develop a Real-time Board Support Package for the Target

With the target microcontroller platform selected, an underlying set of code needed to be developed to support the VM. This included selecting a compiler and debug tools to develop the code. The code to be developed needed to boot and initialize the microcontroller and also provide an interface to interrupts, timers, and serial ports so that a proper evaluation could be performed. A console interface then needed to be developed so that the embedded platform could communicate with a host computer to enter commands and observe results.

4.2.3 Phase Three – Port the Virtual Machine for Execution on the Target

The third phase of the research was to modify the selected VM so that it could run on the target platform. Many VMs were designed with specific targets in mind and most were not designed to run on embedded microcontrollers with Harvard architectures because of the limitations imposed by these platforms. Many VMs expect to load bytecodes from a file system into memory and then execute the program out of RAM. But many small platforms have no external file system and the virtual instructions need to be integrated into the static flash memory space. There are challenges in getting a VM compiler to generate bytecodes that can then be loaded into the flash memory of a microcontroller for later execution. Also, many VMs are designed so that the interpreter is a standalone application rather than one component in a real-time software system. Additionally, most small microcontrollers are limited in the number of languages that are available for compiling native code with C or C++ as the primary language (IAR Systems, 2010) (Byte Craft Limited, 2010). This could have been an issue if the interpreter could not easily be recompiled to run on the target microcontroller. Porting the interpreter to the microcontroller platform was also a part of this second phase. Once the interpreter was ported, it needed to be tested to verify that the conversion was done properly. For example, for the Pascal

language this meant testing the resulting interpreter against the ISO 7185 test suite (ISO 7185, 1990).

4.2.4 Phase Four – Evaluate and Optimize the Virtual Machine

The fourth phase of the research was to evaluate the performance of VM applications running on the microcontroller platform and then to apply some of the optimizations discussed in previous research. The incremental improvements gained with these optimizations were measured and evaluated to determine the usefulness of running the VM on that target platform.

4.2.5 Phase Five – Extend the Virtual Machine with High Level Abstraction

The fifth and final phase of the research involved extending the language and VM to support new capabilities to test the second hypothesis. This is another form of optimization; namely, providing a layer of abstraction so that the capabilities of the underlying platform can be exercised and controlled by the VM. New high level language extensions were developed to manage low level real-time operations so that the advantages of the VM and of the underlying real-time system could be demonstrated working together.

5 The Results of the Study

5.1 Phase 1A Results: Evaluating Candidate Small Embedded Microcontrollers

By definition, this research is focused on the small embedded microcontroller platform. A small embedded platform is quite different from large embedded platforms. A large embedded platform generally has a high performance processor with significant amounts of RAM and code space. These are commonly found in smart phones and high performance embedded systems such as routers and switches. Large embedded platforms are not the focus of

this research because these platforms have already been proven to be able to run VMs such as Java, LLVM and Dalvik. Similarly, there are tiny embedded processors that have a few kilobytes of RAM and code space. Although there has been research with these targets, they are not a part of this research because they are limited to special purpose VMs with 20 or so specialized instructions (Levis & Culler, 2002).

Selection of a microcontroller for this research was done using the criteria specified for a *small* platform. As previously mentioned this required that the device be an 8 or 16 bit microcontroller that operates between 1 and 20 DMIPS, addressed no more than 256K of code space and another 128K of RAM. Table 1-1 lists various microcontrollers of which the Atmel ATxMega family and the PIC24 fit the role of a small microcontroller. Atmel provides the ATxMega128A1 on an inexpensive evaluation platform that is completely self-contained and very suitable for this research. The Atmel XPlain kit provides the ATxMega device on a board along with 8 megabytes of RAM which is more than enough to perform the research (Atmel XPlain, 2010). As shown in Figure 3-1, it also includes a breakout of port pins, a serial port to USB conversion, and a JTAG interface for programming and debug. The board operates on an internal 2 MHz clock which is under speed for the research but the board was retrofitted with a readily available external 12MHz crystal which brings the speed of the microcontroller up near the bottom end of the definition, running at 0.87 DMIPS. Although this is slightly below the lower limit of the definition, the processor supports clock rates up to 32 MHz and the research can be linearly extrapolated to fit within the proper limits.

Tools are also readily available for developing the software to run on the platform. The IAR Embedded Workbench compiler integrated development environment (IDE) for the Atmel AVR family was obtained for coding in C, C++, and assembly (IAR Systems, 2010).

Additionally, a compatible JTAG MKII emulator was obtained which can burn code into the microcontroller's flash memory and can be used in conjunction with the IDE to set breakpoints and single step through the code.

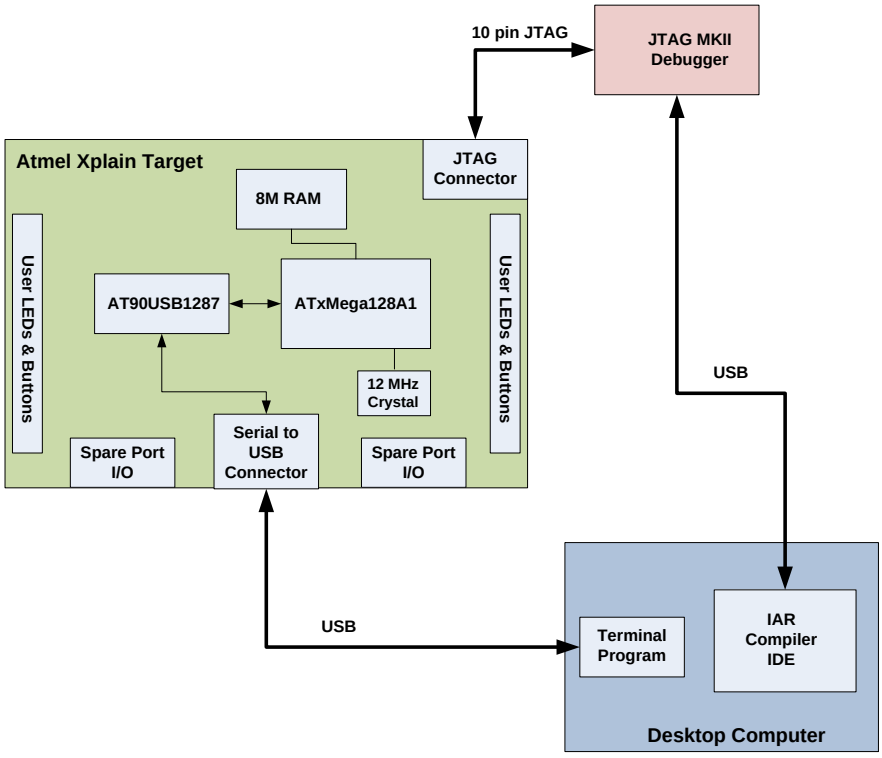

Figure 3-1 Research Development and Test Configuration

5.2 Phase 1B Results: Evaluating Candidate Virtual Machines for Small Targets

Dozens of VMs have been developed in recent years with a broad range of characteristics and there are a number of criteria for deciding on a VM to use. A major issue is finding a VM that will fit into the memory footprint of the target device. For small embedded platforms the target footprint turns out to be the most important factor because most VMs are far too large to fit on a small platform. If the target platform has 128K available for code, it is reasonable to expect that the VM should only occupy a fraction of that space. This constraint is because room

is needed for underlying board support code as well as a significant amount of room for the bytecodes of the application itself. Some platforms have external storage that could be used to store bytecodes and then the application could be loaded at boot up into RAM. But this fact cannot be assumed for many small platforms where the code space in flash memory is all that is available.

An analysis of some existing candidate virtual machines was made to determine the target code footprint size. Some VMs advertise an actual footprint size while others do not supply this information and can only be estimated by counting source lines of code (SLOC). Many popular application VMs and their estimated target footprint are listed in Table 3-1.

Virtual Machine	**Version**	**Estimated Code Footprint**	**Calculation Method**
Java SE	6	68 MB	Cited in literature (Oracle, 2010)
Java SE Embedded "Small Footprint"	6	29.5 MB	Cited in literature (Oracle, 2010)
Java J2ME – CDLC	1.1	128 KB	Cited in literature (Topley, 2002)
Java Card	3	256 KB	Cited in literature (Oracle, 2008)
.NET Compact Framework	2.0	5.5 MB	Cited in literature (Microsoft, 2010)
.NET CLI (Mono)	1.1	4 MB	Cited in literature (Novell, 2010)
.NET Micro Framework	4.0	200 KB	Cited in literature (Micro Framework, 2010)
Dis (Inferno)	4.0	231 KB	Estimate, SLOC = 28915 (Dis Virtual Machine Specification, 1999)
Parrot	2.2.0	240 KB	Estimate, SLOC=30019 (Parrot, 2010)
LLVM	2.6	996 KB	Estimate, SLOC= 124458 (LLVM, 2010)
LUA	5.1.4	81 KB	Estimate, SLOC= 10152 (Lua the Programming Language, 2010)
eLUA	0.7	163 KB (includes BSP)	Estimate, SLOC = 20386 (eLua - Embedded Lua, 2010)
Squirrel	3	79 KB	Estimate, SLOC = 9866 (Demichelis, 2010)
PICOBIT (Scheme)	N/A	15.6 KB	Cited in literature (St-Amour & Feeley, 2009)
P-Code (Standard Pascal)	P5	18 KB	Estimate, SLOC = 2255 (Moore, 2010)
M-Code (Modula-2)	M2	16KB	Measured from DOS exe (Modula-2, 2010)

Table 3-1 – Candidate Application Virtual Machines

When the small embedded target is defined to have 128KB and at most 256K of code space, most existing VMs are eliminated. The Java 2 Micro Edition (J2ME) Connected Device Limited Configuration (CDLC) is advertised to require a minimum of 128K ROM and 32K RAM to contain the bare VM (Topley, 2002). It recommends 512K total memory available for the VM. On the surface this seems like it might work. But when one considers that there needs to be room for an underlying real-time native system, even this candidate exceeds the limits of a small embedded microcontroller platform. Similarly, .NET Micro Framework requires a minimum of 200K of code space and has been ported to run on many mid-range embedded platforms in such implementations as TinyCLR (GHI Electronics, 2012). But in the small microcontroller with <=256K of code space there is very little room left for application space.

From the above list only LUA, eLUA, Squirrel, PICOBIT, Modula-2, and Pascal variants are left as possible candidates. Eliminating the dynamic scripting languages from this list leaves only the venerable Pascal and Modula-2 virtual machines. There are a number of other lesser known virtual machines that have been created that might also be considered. Many of them are from prior research efforts (Craig, 2005). These generally are limited by the demonstration language provided with them and are not ready for general purpose use. It is not the purpose of this research to exhaust all possible candidates, but rather to find a representative VM that meets minimum criteria and evaluate its performance on the selected target. Based on open source tool availability and language capabilities, for this research the Standard Pascal P5 VM was selected. Despite its age, Pascal has been maintained and updated over the years and exists in a modern form with ISO standardization and a test suite to insure compliance (ISO 7185, 1990). The source code for the compiler and the interpreter are readily available and it implements a full

general purpose language. Additionally, the VM easily fits within the footprint of a common small microcontroller.

The tools that are available for P5 include the source code to both the compiler and the interpreter, both written in Pascal. The P5 platform is currently supported and maintained. The compiler and interpreter can both be hosted on a Windows PC by using the GNU Pascal compiler to compile the sources (Virtanen, 2005). This has been done and both were successfully compiled into executable binaries with no difficulty.

5.3 Phase 2 Results: Develop a Real-time Board Support Package for the Target

In support of the research, a board support package was developed for the Atmel XPlain target and the ATxMega128A1 microcontroller. The IAR Embedded Workbench and JTAG MKII debugger were used to create and download the code to the target. The platform can boot and initialize the board including the I/O ports, interrupt system, serial ports, and timers. It also has a custom dynamic memory manager which was used later in the research by the virtual machine as a part of the messaging system.

5.4 Overview for Phases Three, Four, and Five

The next three phases of the research involved porting the Pascal P5 VM to the target platform and then making modifications to the compiler, assembler, and the interpreter to evaluate the hypotheses. Figure 3-2 illustrates the steps that were performed in phase three, four, and five.

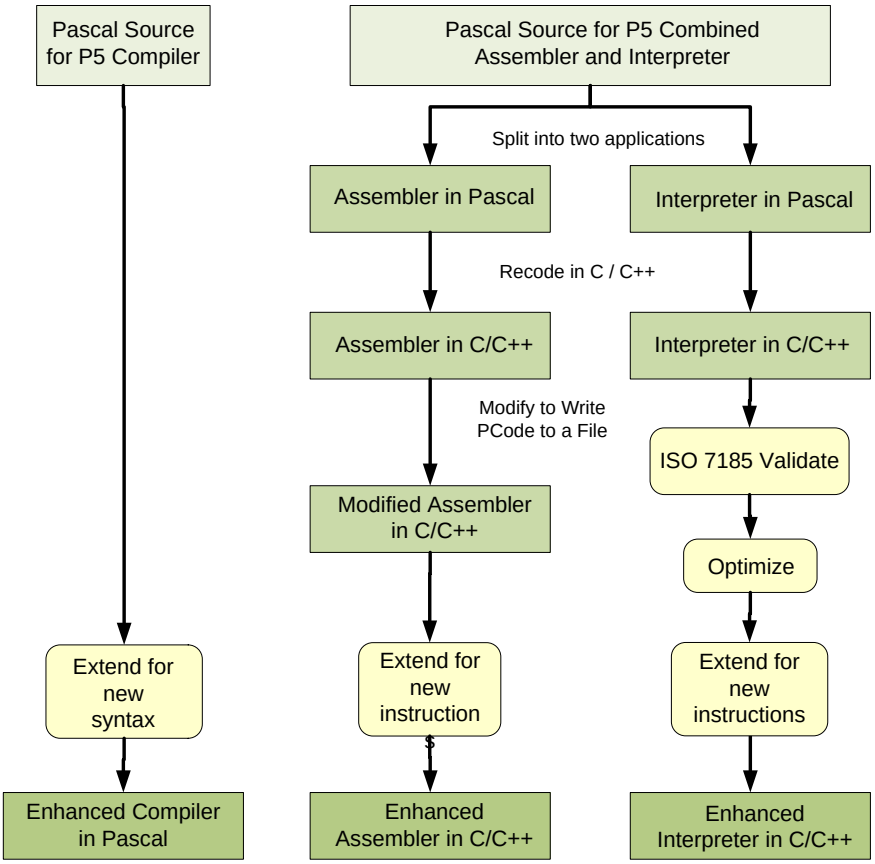

Figure 3-2 Modifications Necessary to Complete Phases Three, Four, and Five

5.5 Phase 3 Results - Modify the Virtual Machine for Execution on the Target Platform

5.5.1 How the Virtual Machine Originally Worked

Several weeks of work were required to port the Pascal P5 VM to run on the target platform. As shown in Figure 3-3, the Pascal P5 system which normally runs on a desktop computer works in two phases (Moore, 2010). First, the compiler converts Pascal source code into an intermediate representation that is stored as a file. Then, to run the program, the combined assembler / interpreter loads the intermediate code file from disk into memory. The assembler converts the intermediate code into bytecodes which remain in an array in RAM and then the interpreter executes the bytecodes to run the program as illustrated below.

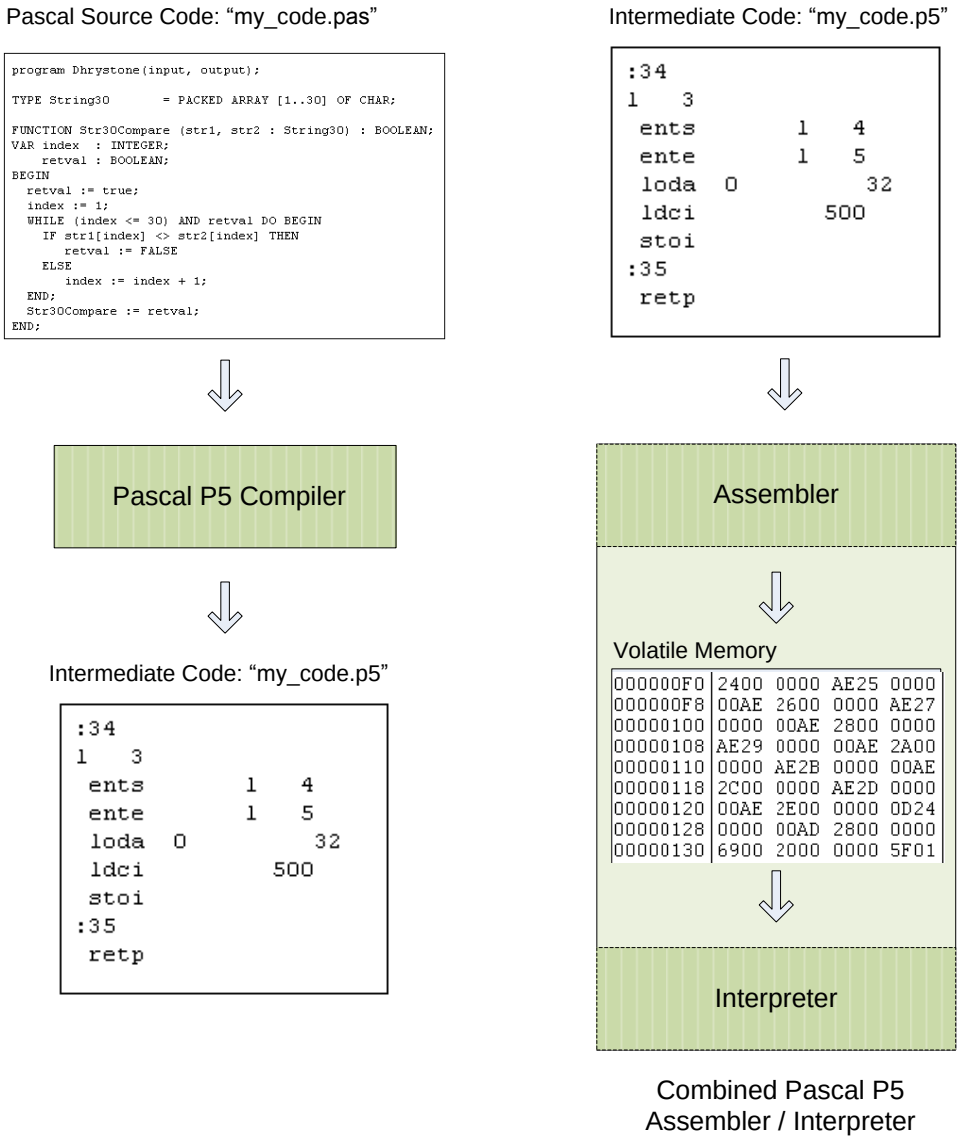

Figure 3-3 Original Pascal P5 Operation

5.5.2 How the Virtual Machine was Modified to Work

The original paradigm of assembling the bytecodes into RAM and then executing them would be very impractical on an embedded microcontroller target which has limited resources. It is far more efficient to assemble the code ahead of time than every time the application is run. As shown in Figure 3-4, the embedded platform requires that the compiler and assembler run on

a development platform and output bytecodes ahead of time. These bytecodes then need to be linked with the board support code for the embedded target as illustrated.

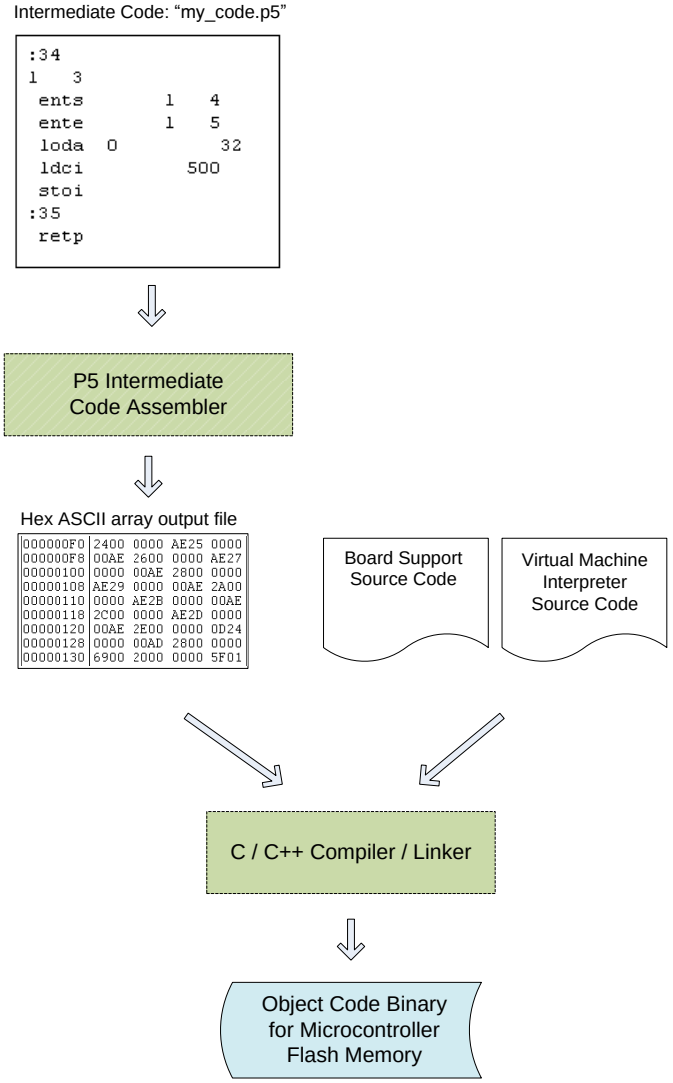

Figure 3-4 Modified Pascal P5 Operation

The assembler / interpreter program for P5 Pascal was supplied in Pascal. This source code was first split into a separate assembler and interpreter and made to work separately. Then the assembler was recoded in Visual Studio C++ 2005 (Microsoft Visual Studio, 2005). This step was done for convenience since the tools for supporting the code with Visual Studio were better than those available for Pascal. The interpreter was then recoded into C/C++ within

Visual Studio 2005 as an intermediate step for ultimately compiling it onto the target platform for execution there. This step allowed the porting of the assembler and interpreter for testing on a PC platform, which was accomplished using the ISO 7185 test application that exercised all of the features of the standard Pascal language (ISO 7185, 1990). The Pascal compiler generated PCode assembly which was then assembled using the new C/C++ version of the assembler, which generated a binary image of the PCode. This PCode binary image was then loaded into the C/C++ version of the interpreter on the PC and executed. The PC platform was thus used to smooth the porting of the assembler and the interpreter.

Once the ISO 7185 application ran successfully, efforts focused on making the tools generate an image that could be run on the small microcontroller target. First, the assembler was extended to output an image that could be easily used on the target platform. The binary image that it was generating was difficult to link into the target, so the assembler was modified to output an addition source code version of the application PCode image. This step was accomplished by generating source code arrays of bytes that the target compiler could easily compile into binary and the executable could then load at startup. An example is shown in Figure 3-5 where there are three arrays, one for the header information, one for the program constants, and one for the PCode program bytes.

```
// source: tasktest.p5

__hugeflash unsigned char pcodeHeader[32] = {
0xFF, 0x7F, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x50, 0x04, 0x00, 0x00, 0x88, 0x7F, 0x00, 0x00,
0x88, 0x7F, 0x00, 0x00, 0x50, 0x04, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x50, 0x04, 0x00, 0x00
};

__hugeflash unsigned char pcode[1104] = {
0x0B, 0x00, 0x0C, 0x00, 0xD0, 0x03, 0x00, 0x00, 0x3A, 0xAE, 0x01, 0x00, 0x00, 0x00, 0xAE, 0x02,
0x00, 0x00, 0x00, 0xAE, 0x03, 0x00, 0x00, 0x00, 0xAE, 0x04, 0x00, 0x00, 0x00, 0xAE, 0x05, 0x00,
0x00, 0x00, 0xAE, 0x06, 0x00, 0x00, 0x00, 0xAE, 0x07, 0x00, 0x00, 0x00, 0xAE, 0x08, 0x00, 0x00,
0x00, 0xAE, 0x09, 0x00, 0x00, 0x00, 0xAE, 0x0A, 0x00, 0x00, 0x00, 0xAE, 0x0B, 0x00, 0x00, 0x00,
0xAE, 0x0C, 0x00, 0x00, 0x00, 0xAE, 0x0D, 0x00, 0x00, 0x00, 0xAE, 0x0E, 0x00, 0x00, 0x00, 0xAE,
0x0F, 0x00, 0x00, 0x00, 0x0D, 0x24, 0x00, 0x00, 0x00, 0xAD, 0x22, 0x00, 0x00, 0x00, 0x69, 0x00,
0x20, 0x00, 0x00, 0x00, 0x0F, 0x02, 0x00, 0x00, 0x00, 0x06, 0xAE, 0x10, 0x00, 0x00, 0x00, 0x0E,
0xAE, 0x11, 0x00, 0x00, 0x00, 0xAE, 0x12, 0x00, 0x00, 0x00, 0xAE, 0x13, 0x00, 0x00, 0x00, 0xAE,

       … condensed for formatting by omitting many rows here …

0x00, 0xAE, 0x43, 0x00, 0x00, 0x00, 0xAE, 0x44, 0x00, 0x00, 0x00, 0xAE, 0x45, 0x00, 0x00, 0x00,
0x0D, 0x28, 0x00, 0x00, 0x00, 0xAD, 0x24, 0x00, 0x00, 0x00, 0x38, 0x92, 0x7F, 0x00, 0x00, 0x04,
0x00, 0x22, 0x00, 0x00, 0x00, 0x76, 0x04, 0x00, 0x00, 0x00, 0x7B, 0x11, 0x00, 0x00, 0x00, 0x7B,
0x11, 0x00, 0x00, 0x00, 0x0F, 0x06, 0x00, 0x00, 0x00, 0x7B, 0x04, 0x00, 0x00, 0x00, 0x7B, 0x0B,
0x00, 0x00, 0x00, 0x0F, 0x08, 0x00, 0x00, 0x00, 0x38, 0x8B, 0x7F, 0x00, 0x00, 0x7B, 0x07, 0x00,
0x00, 0x00, 0x7B, 0x07, 0x00, 0x00, 0x00, 0x0F, 0x06, 0x00, 0x00, 0x00, 0x7B, 0x88, 0x13, 0x00,
0x00, 0x7B, 0x0B, 0x00, 0x00, 0x00, 0x0F, 0x08, 0x00, 0x00, 0x00, 0x0F, 0x05, 0x00, 0x00, 0x00,
0x75, 0x04, 0x00, 0x00, 0x00, 0xAE, 0x46, 0x00, 0x00, 0x00, 0xAE, 0x47, 0x00, 0x00, 0x00, 0x0B,
0x00, 0x0C, 0x00, 0xC5, 0x00, 0x00, 0x00, 0xAE, 0x48, 0x00, 0x00, 0x00, 0x0E, 0x00, 0x00, 0x00
};

__hugeflash unsigned char pcodeConst[119] = {
0x00, 0x00, 0x00, 0x23, 0x20, 0x6C, 0x6F, 0x6F, 0x70, 0x73, 0x53, 0x74, 0x61, 0x72, 0x74, 0x69,
0x6E, 0x67, 0x2C, 0x20, 0x23, 0x20, 0x74, 0x61, 0x73, 0x6B, 0x73, 0x20, 0x6D, 0x69, 0x6C, 0x6C,
0x69, 0x73, 0x65, 0x63, 0x6F, 0x6E, 0x64, 0x73, 0x4C, 0x6F, 0x6F, 0x70, 0x20, 0x54, 0x69, 0x6D,
0x65, 0x20, 0x3D, 0x20, 0x20, 0x6D, 0x69, 0x6C, 0x6C, 0x69, 0x73, 0x65, 0x63, 0x6F, 0x6E, 0x64,
0x73, 0x54, 0x61, 0x73, 0x6B, 0x20, 0x53, 0x77, 0x69, 0x74, 0x63, 0x68, 0x20, 0x2B, 0x20, 0x4C,
0x6F, 0x6F, 0x70, 0x20, 0x54, 0x69, 0x6D, 0x65, 0x20, 0x3D, 0x20, 0x66, 0x6C, 0x61, 0x67, 0x20,
0x3D, 0x20, 0x50, 0x61, 0x72, 0x65, 0x6E, 0x74, 0x20, 0x63, 0x72, 0x65, 0x61, 0x74, 0x69, 0x6E,
0x67, 0x20, 0x74, 0x61, 0x73, 0x6B, 0x20  };
```

Figure 3-5 Output of PCode Header, Code, and Const Sections for Target Platform

Finally, the C/C++ version of the interpreter that was running under Visual Studio on a PC was then ported to the target platform by recompiling the source code with the IAR Embedded Workbench for the Atmel AVR platform (IAR Systems, 2010). The interpreter user I/O was redirected to use the platform board support package I/O so that user interaction would occur through the serial port console. Thus the read/write standard I/O of the Pascal application was redirected to show up on the HyperTerminal session on the development PC. To test the interpreter on the target platform, the same ISO 7185 Pascal application was then assembled into the new source output format. Then this C/C++ source code output was compiled with the rest of the target platform code. By running the ISO 7185 application on the target platform, any

errors in the interpreter were found and resolved. At this point the complete development platform was ready for any Pascal program that would be written for running on the small microcontroller platform.

5.6 Phase Four Details – Evaluate and Optimize the Virtual Machine

5.6.1 Extending the Compiler, Assembler, and Interpreter for a Time Function

Before any performance evaluation can be performed on a platform, there must be a method for measuring time. The native environment has a background timer that is based on the crystal and the 10 millisecond tick. When the board support package was developed, support for reading this timer was added which allows the native code to time itself.

The Pascal interpreter does not have a built-in function for reading time which makes it difficult to measure performance while running in the virtual domain. As a result, the Pascal compiler was extended to add a new intrinsic function called *time( )* which returns a simple integer value indicating the current reading of the 10 millisecond board support package (BSP) background tick. This function allows Pascal code to obtain a start and a stop time for calculating elapsed time when running performance measurements. The Pascal compiler was modified to generate a new intermediate code *time* instruction and the assembler was modified to accept the intermediate *time* instruction from the compiler and then generate a corresponding bytecode instruction. Finally, the interpreter was extended to process this new bytecode instruction. The interpreter calls into the native board support code and captures the timer tick and puts it on the stack for the subsequent instruction to load into a variable (Figure 3-6).

```
case 2: // time = put the time in ticks on the stack
  {
    INT32U         currentTicks;
    UINT8          currentTicksExt;
```

```
  GetTimer0Count(&currentTicks,&currentTicksExt);
  pshint(currentTicks);
}
break;  // case 2
```

Figure 3-6 Interpreter Extension to Support Time( ) Function

5.6.2 Evaluating Virtual Machine Performance using a Dhrystone Measurement

One method to evaluate the performance of a VM is to compare its execution against an equivalent operation in native code. Performance was measured in phase four by benchmarking the performance using the Dhrystone mix (Weicker, Dhrystone: a synthetic systems programming benchmark, 1984). The Dhrystone test measures integer performance over a range of typical program statements including function calls, loops, condition statements, math operations, and array operations. The Dhrystone measurement has been shown to have limitations but has been shown still to be a useful measurement for small embedded microcontrollers (Weiss, 2002). The Dhrystone measurement was updated to version 2 in 1988 to correct a problem where optimizing compilers could artificially appear to improve the performance of the processor (Weicker, 1988). This measurement was first made by running the Dhrystone test in compiled native C code on the platform to use as a baseline for the theoretical limit. Versions of the Dhrystone test in the C language are readily available on the internet (Longbottom, 2010). Dhrystone source code was compiled using the native C compiler and then it was run on the target platform and the performance was measured and recorded in Table 3-2. Once the native timing was measured the attention then turned to running the same test on the target in the virtual domain. Ordinarily, a Pascal version of the Dhrystone test would need to be compiled using the Pascal P5 compiler, but no such version existed, However, a freeware Dhrystone test and associated compiler in Modula-2 was found (XDS Modula-2 Compiler). Because Modula-2 is very similar to Pascal it was not difficult to translate the code to Pascal.

The resulting source code was then compared line by line with the C version to insure that the two perform the same operations. This Pascal source code version of the Dhrystone test was then compiled using the modified Pascal P5 compiler and the resulting PCode was used for benchmarking the baseline VM and each subsequent optimization. The results of the comparison between native and virtual execution are shown in Table 3-2. Additionally, execution on a standard modern desktop PC is also shown to contrast the typical speed of a small microcontroller compared to a PC. As shown in the table, in this test a modern PC ran over 9,400 times faster than the microcontroller. This effect is consistent with the magnitude of the problem where a PC can effectively run many different VMs, but a small microcontroller is very limited in performance. Furthermore, without optimization the VM version of the Dhrystone test ran more than 500 times slower than the native equivalent on the PC. On the small microcontroller, the VM version of the Dhrystone test ran about 275 times slower than the native equivalent. The difference between ratios on the two platforms is because the PC version used an interpreter that came with the Pascal P5 baseline, whereas the microcontroller interpreter was recoded in C as part of the porting process. The original Pascal interpreter was likely compiled using a less-efficient Pascal x86 compiler. Additionally, the PC contains a very large cache which helps small native applications because they can locate themselves entirely in cache, exaggerating the speed at which they can operate.

Platform	Speed	Dhrystones/ second	DMIPS
PC Native	3.2 GHz (Intel i5 650)	14,428,241	8211.86
PC Virtual	3.2 GHz (Intel i5 650)	28,571	16.26
ATxMega AVR Native	12 MHz	1,529	0.87
ATxMega AVR Virtual	12 MHz	5.562	0.003165

Table 3-2 Unoptimized Virtual Machine Speed Comparison

5.6.3 Optimizing the Virtual Machine

The next step in the process was to optimize the VM to speed up the execution. This is done before testing the first hypothesis of whether or not it is practical to run a VM on a small embedded microcontroller. This is so that the hypothesis will be tested against a fast version rather than an unoptimized one. Optimizations were made incrementally, and with each one the Dhrystone test was rerun and measured. The results were added to Table 3-3 which shows the incremental improvement measurements.

5.6.3.1 Optimize Register Fetch Routines

The first optimization was to convert some critical code to be inline, trading code size for speed of execution. The Pascal VM instruction set has very few virtual registers. The primary ones are OP, P, and Q. A secondary Q, called Q1 was also added to the instruction set for array operations. The original interpreter code made a function call to obtain each of these values out of the bytecodes for each instruction that used the registers. The optimization was to replace each of these calls with inline code as shown in Figure 3-7. This simple change resulted in about a 10% speed improvement but cost over 20% more code space.

```
#ifdef OPTIMIZE_UNROLL_GETQ
  #define getq()  Qreg = getadr(PCreg);  PCreg += adrsize;
#else
  void getq (void)
  {
    Qreg = getadr(PCreg);
    PCreg += adrsize;
  }  // getq
#endif
```

Figure 3-7 Inlining of Register Fetch Routines

5.6.3.2 Optimize Address Get and Put Routines

The next optimization involved the routines that get and put addresses on the stack. These two routines are called frequently in the interpreter and the original Pascal code that was

translated into C was not written to take advantage of implicit compiler optimization. The optimization as shown in Figure 3-8 resulted in nearly an additional 14% speed improvement with no additional code space required. The improvement is largely because the compiler is able to use the native instruction set to build and use pointer values more efficiently than can be done manually in C.

```
address getadr (address a)
{
#ifdef OPTIMIZE_GETADR
#if (adrsize != 4)
#error OPTIMIZE_GETADR Failed
#endif

  UINT32 *pAdr = (UINT32 *) &(store[a]);
return(*pAdr);

#else
union {
address u;
    BYTE    b[4];
} r;

  for (int i=0; i<adrsize; i++)
  r.b[i] = store[a+i];
return(r.u);
#endif
}  // getadr
```

Figure 3-8 Optimizing the getadr() Routine

5.6.3.3 Optimize Variable Get and Put Routines

In addition to optimizing the register and address routines, the routines that get and put integer, boolean, and character variables are also subject to optimization. The technique for optimizing these routines is similar to the address get and put routines. As shown in Table 3-3, this optimization resulted in nearly 18% more speed improvement and actually resulted in slightly smaller code.

5.6.3.4 Optimize Main Switch Statement

An optimization that failed was an attempt to speed up the main case statement that processes the virtual opcodes. The original code uses a large C-code "switch" statement where each case contains the code to process a single opcode as shown in Figure 3-9.

```
while (!done) {
getop();    // fetch the next opcode

switch (OP)  {
case 0:   // lodi
getp(); getq(); pshint(getint(base(Preg) + Qreg));
    break;
  case 105: // loda
  getp(); getq(); pshadr(getadr(base(Preg) + Qreg));
break;

  // the rest of the cases go here

} // switch statement
} // while not done
```

Figure 3-9 Original Switch Statement

This code was modified to use a jump table as shown in Figure 3-10. The results of the measurement demonstrate that the overhead of making a subroutine call for each instruction costs more than the value of avoiding the switch statement. As shown in Table 3-3 this resulted in 38% slower operation. As a result the jump table was taken back out of the code and the switch statement was restored.

```
typedef void (*FUNCPTR)(void);

while (!done) {
getop();    // fetch the next opcodes

// call the function directly
(jumpTable[OP])();

} // while not done

FUNCPTR jumpTable[256] = {
OPCODE_000, OPCODE_001, OPCODE_002, OPCODE_003, OPCODE_004,
 OPCODE_005, OPCODE_006, OPCODE_007, OPCODE_xxx, OPCODE_009,
OPCODE_010, OPCODE_011, OPCODE_012, OPCODE_013, OPCODE_014,
OPCODE_015, OPCODE_016, OPCODE_017, OPCODE_018, OPCODE_019,

  // the rest of the table goes here
};

void OPCODE_000(void) {   // lodi
    getp(); getq(); pshint(getint(base(Preg) + Qreg));
}
```

Figure 3-10 Switch Statement Converted to a Jump Table

Platform / Optimization	Native / Virtual Ratio	Dhrystones per sec / DMIPS	Code Size (bytes)
Native	1:1	1,529 / 0.87	4,573 native
Virtual, No Optimization	275 : 1	5.562 / 0.003165	22,697 VM 6,063 bytecodes
Unroll register fetch	251 : 1	6.096 / 0.003469	27,391 VM 6,063 bytecodes
improve get/put address	221 : 1	6.934 / 0.003946	27,391 VM 6,063 bytecodes
improve get/put int,char,bool	187:1	8.162 / 0.004645	27,331 VM 6,063 bytecodes
Function jump table	303:1	5.051 / 0.002874	29,555VM 6,063 bytecodes

Table 3-3 Virtual Machine Speed Improvement Attempts

5.6.4 Evaluating the Virtual Machine Performance

A typical use of a small microcontroller is for performing a set of operations that include certain real-time operations. These operations may include such things as monitoring ports for human interaction such as key presses and display updates. It may also include interacting with

modestly high speed peripherals such as analog to digital conversion and synchronous and asynchronous serial communications (Atmel Product Guide, 2011). To evaluate the suitability of the VM, experiments were devised that involved asynchronous serial communication and measured the performance of the platform. The experiments involved looping back an unused serial port on the processor so that data that was sent out was immediately received back into the processor. Then the port was operated at various standard data rates as data was transmitted and received at the maximum rate that the VM could support. A fixed number of characters of data were looped through this interface and the time was measured and compared against the maximum theoretical limit. This provided a percentage measurement of actual versus theoretical throughput.

When the VM interfaces with actual hardware, it does so by supporting byte codes that call into the native software and hardware platform to perform the operations. To perform this experiment, the language was extended to support a new *serial* procedure.

For example, to initialize the port and set the data rate to 9600 bps (bits per second):

```
serial(SERIALINIT, 9600, nil);
```

To transmit a buffer of data, one could load an array with the data, and call the function as follows:

```
FOR xmit := 1 TO XmitBufSize DO
BEGIN
  xmitbuffer[xmit] := dataToSend[xmit];
END;

serial(SERIALXMIT, xmit, xmitbuffer);
```

Then, to receive some data, one could call the function and supply a buffer. The second parameter is a variable parameter that indicates how big the buffer is. The value gets changed by the VM upon return from the function to indicate how many characters of received data were

actually loaded into the buffer. This allows the serial procedure to return with whatever data is pending without blocking to wait until the buffer is full, allowing the program to do other work while waiting for characters to arrive and for the interrupt routine to put them into the receive queue:

```
REPEAT
  rcv := RCVBUFSIZE;
  serial(SERIALRCV,  rcv, rcvbuffer);
  totalRcvd := totalRcvd + rcv;
UNTIL (rcv < RCVBUFSIZE);
```

On the receive side, the bytes were removed from the packet one at a time and compared against the expected result to simulate a more realistic evaluation of received data. Table 3-4 and Figure 3-11 show the results. In this case, single byte processing begins to fall behind at 1200 bps rather than at 2400 bps like in raw mode. Processing of larger packets begins to fall behind at 2400 bps for both 10 and 25 byte packets rather than at 19,200 bps. Additionally, there was virtually no improvement going from 10 to 25 bytes. This could be explained by the fact that the improvement generated by increased packet size was nearly overshadowed by the need to process each byte in the larger packet using the VM. Thus it would seem that larger packet sizes lose their advantage in a VM, and the VM performance becomes bounded by the data rate alone. It may also be a warning against attempting to do too much character-level processing at the VM level. Instead, a VM should be designed where that processing is somehow handled in native code. For example, if a CRC (cyclic redundancy check) needed to be performed on the packet, it would be wise to perform this function in native code as data is received or as a part of a superoperator bytecode that calculates a CRC all at once within the VM. Any other character-level processing would also best be handled as a packet all at once within a single instruction in the VM rather than one character at a time.

5.6.4.1 Evaluation of Serial Port Test Results

These serial port tests were performed on a platform that runs at about 0.87 DMIPS. The enhanced serial port test demonstrated that the VM on this platform can maintain a fair performance up to about 2400 bps. If very little processing is required in real-time, the VM on this platform can perform well up to about 19200 bps. A small microcontroller platform was defined as one that runs up to about 20 DMIPS. By linearly extrapolating up to 20 DMIPS, the high end of the *small* range should manage well up to 57600 bps. A VM running on a small microcontroller platform can manage and process real-time full duplex serial data effectively at 50% or better efficiency from 2400 bps to 57600 bps depending on where the platform is in the range between 1 and 20 DMIPS. Even at 115200 bps a high end platform will run at about 28% efficiency. This efficiency is aided by the underlying native code that manages real-time interrupts and buffering.

Data Rate	300	600	1200	2400	4800	9600	19200	38400	57600	115200
Packet Size	1	1	1	1	1	1	1	1	1	1
# Bits	10000	10000	10000	10000	10000	10000	10000	10000	10000	10000
Actual Time (ms)	33400	16710	13410	11440	13420	11480	11440	11440	11440	11440
Theoretical Time (ms)	33333	16667	8333	4167	2083	1042	521	260	174	87
% Efficient	99.80	99.74	62.14	36.42	15.52	9.07	4.55	2.28	1.52	0.76

Data Rate	300	600	1200	2400	4800	9600	19200	38400	57600	115200
Packet Size	10	10	10	10	10	10	10	10	10	10
# Bits	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000
Actual Time (ms)	333910	166980	83520	70360	82370	70360	70730	71570	71570	71560
Theoretical Time (ms)	333333	166667	83333	41667	20833	10420	5208	2604	1736	868
% Efficient	99.83	99.81	99.78	59.22	25.29	14.80	7.36	3.64	2.43	1.21

Data Rate	300	600	1200	2400	4800	9600	19200	38400	57600	115200
Packet Size	25	25	25	25	25	25	25	25	25	25
# Bits	250000	250000	250000	250000	250000	250000	250000	250000	250000	250000
Actual Time (ms)	834760	417430	208760	167990	196610	167990	168360	171770	171770	171770
Theoretical Time (ms)	833333	416667	208333	104167	52083	26042	13021	6510	4340	2170
% Efficient	99.83	99.82	99.80	62.01	26.49	15.50	7.73	3.79	2.53	1.26

Table 3-4 Serial Port Test Raw Data

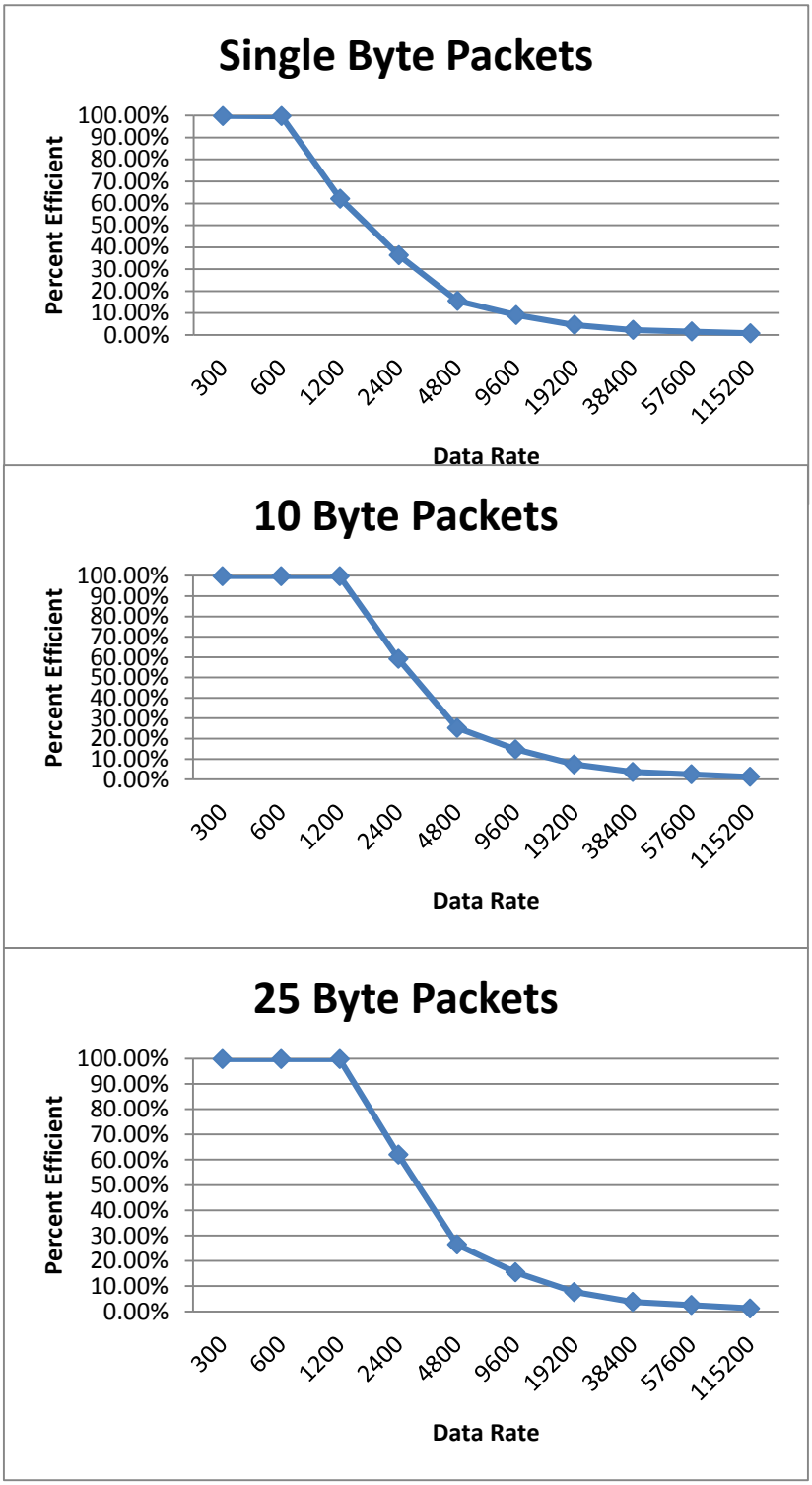

Figure 3-11 Serial Port Test Results

5.7 Phase 5 Results – Extend the Virtual Machine with High Level Abstraction

A different approach to extending the VM was taken in phase five. The first research was to determine if a suitable VM and language could be found that would be suitable to run on a small microcontroller target. The second part was to determine if the VM could be extended to run effectively on the target through the use of abstraction. This is important because research has shown that even a highly optimized VM can run magnitudes slower than its native counterpart (Kazi, Chen, Stanley, & Lilja, 2000). As mentioned earlier, NanoVM ran an average of 191 times slower than the native equivalent (Harbaum, 2006). The Pascal VM for this research ran 187 times slower after the optimizations were made. The only way for a VM to be effective is if it adds value to the platform. Besides security and robustness, the primary advantages are code compaction and high level abstraction through a powerful instruction set. This is in keeping with the research that has been done with superoperators, superinstructions, and macro instructions. The problem with the research on superoperators, superinstructions, and macro instructions is that the emphasis was on performing statistical measurements to combine low level instructions into higher level ones that provide limited benefits for a particular application or for a particular mix of instructions. If the mix of instructions changed, some of the benefits would be lost. Additionally, in some cases the interpreter had to be replaced on the target each time a new superoperator was calculated.

Instead of simply combining low level operations into more powerful bytecodes, new language and instruction extensions can be created to support higher level operating system-like services to prove the benefits of abstraction. This follows more along the line of the domain-specific language (DSL) approach where a language and instruction set is customized to meet the needs of an entire class of applications. This customization includes very abstract instructions

that are geared toward the problems being managed. For a small embedded microcontroller, the abstraction should not only include handling real-time events, but also performing abstract high level services efficiently so that the drawbacks of running a VM can be overcome through abstraction. By showing that a merely "suitable" VM can be made to run effectively through abstraction, the second hypothesis would be affirmed.

5.7.1.1 Standard Architectures to Support Concurrency

One common architecture for a small embedded platform is to have a single native thread of operation which manages multiple interrupt sources and several contexts via a cyclic executive that periodically polls each context. As shown in Figure 3-12, a VM could be incorporated into this architecture by adding the interpreter as one of the contexts that would be managed by calling into the interpreter on a regular basis to execute a series of bytecodes. Multiple VM threads could be supported by defining multiple VM contexts that are managed within the main loop.

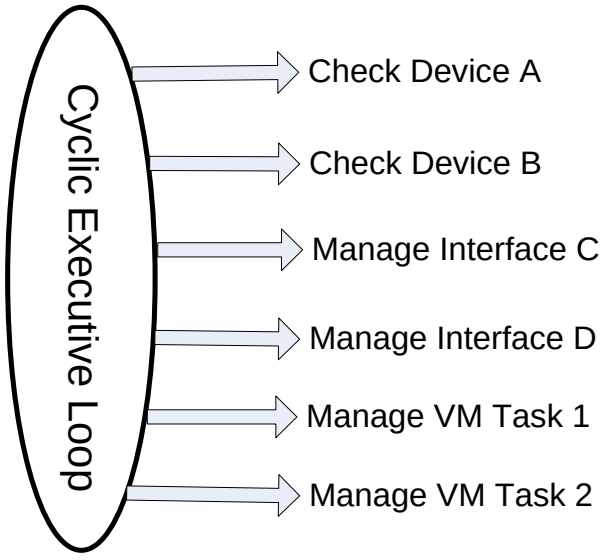

Figure 3-12 Example Single-Threaded Architecture Supporting Multiple VM Tasks

Another common architecture for a small embedded platform is to run a real-time operating system and have multiple tasks that interact via message passing and semaphores (Micrium, 2010). A VM could be added in this architecture by dedicating one of the tasks to run the VM. As shown in Figure 3-13, multiple VM applications could be run by having multiple tasks that are each running a VM context, all under the control of the native real-time operating system.

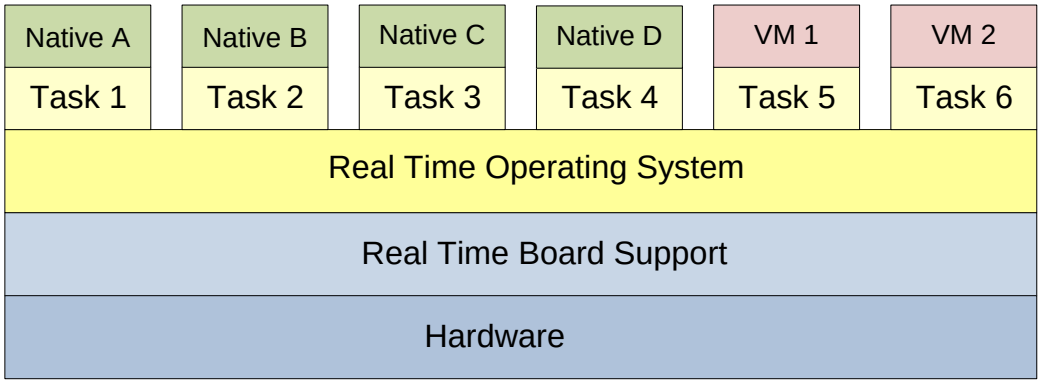

Figure 3-13 Example RTOS Architecture with Multiple VM Tasks

5.7.1.2 Using Virtual Machine Abstraction to Support Concurrency

Instead of using one of these two architectures, a different paradigm for supporting a VM was adopted. An efficient use of a hybrid system of both native and virtual instructions is to allow each to do what it does best. Native code is more efficient at real-time processing and low level calculations than virtual code because a VM is inefficient at executing individual bytecodes because of the time spent in the execution loop (Eller, 2005). This situation was shown to hold true in the serial port tests where character-level processing was done in the native code much more efficiently and the packet level processing was controlled by the VM. Virtual code has an advantage over native code when it comes to high level abstraction because the instructions can be as abstract as the designer makes them without regard to complexity since there is no need to

implement the instruction in hardware. The more abstract the instruction, the fewer the number of low level instructions that are needed. Thus by showing that there are some useful abstract instructions that perform effectively on a target platform, the second hypothesis would be supported.

One abstract extension to a VM that was found to provide great overall value to the system is concurrency. As shown in Figure 3-14, a VM that maintains its own concurrency has advantages over a real-time operating system. Switching contexts in a VM can be done in one virtual instruction whereas a native RTOS-driven system requires saving and restoring contexts which may involve dozens of variables and the saving of a task state. Also, a concurrent VM is more controlled than a real-time operating system task switch which can occur at almost any time requiring critical regions of code to protect against race conditions and errors. There are also run time licenses that often need to be purchased to have an RTOS in the system. When concurrency is performed within the VM environment, then it also made sense to supply abstractions for message passing and task management within the virtual environment. The goal of the extensions was for the VM to manage the platform with multiple threads of execution while leaving the real-time operations and low level processing to occur in the underlying native environment, which permits both the virtual domain and the native domain to do what they each do best.

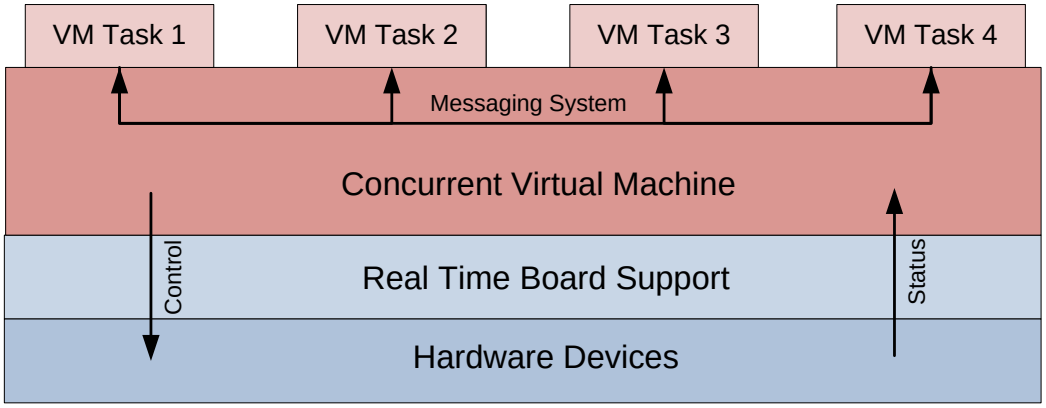

Figure 3-14 Example Alternate Paradigm with Concurrent Virtual Machine

5.7.1.3 Implementing a Task Instruction to Support Concurrency

New instructions were added to the Pascal Virtual Machine to support multitasking and message passing as high level abstract instructions. A new *task* instruction was added to the interpreter to support multitasking along with the necessary Pascal language extension as follows:

```
status := task(function);
 function:
       TASKINIT=0: status: 1=parent, 0=child
       TASKPEND=1: status: is new task ID
       TASKWHO=2:  status: is my task ID
       TASKKILL=3: No status
```

With this new special function call, new tasks could be created using the subfunction TASKINIT which performs a similar function as the UNIX fork( ) command within a single PCode instruction. Modeling task creation after the UNIX fork command was a matter of convenience for this research although it does have some advantages. Child tasks automatically inherit their own copies of the state of the parent task including all of the variables and their current values. The interpreter was modified to support up to 5 tasks (an arbitrary choice that depends on the memory available). When a new task is created, an exact duplicate of the working stack of the parent task is created and a new entry is made in a TaskState table for the

new child task. The return value from this function call indicates whether the currently executing task is still the parent task (1) or the child task (0) that is an exact copy.

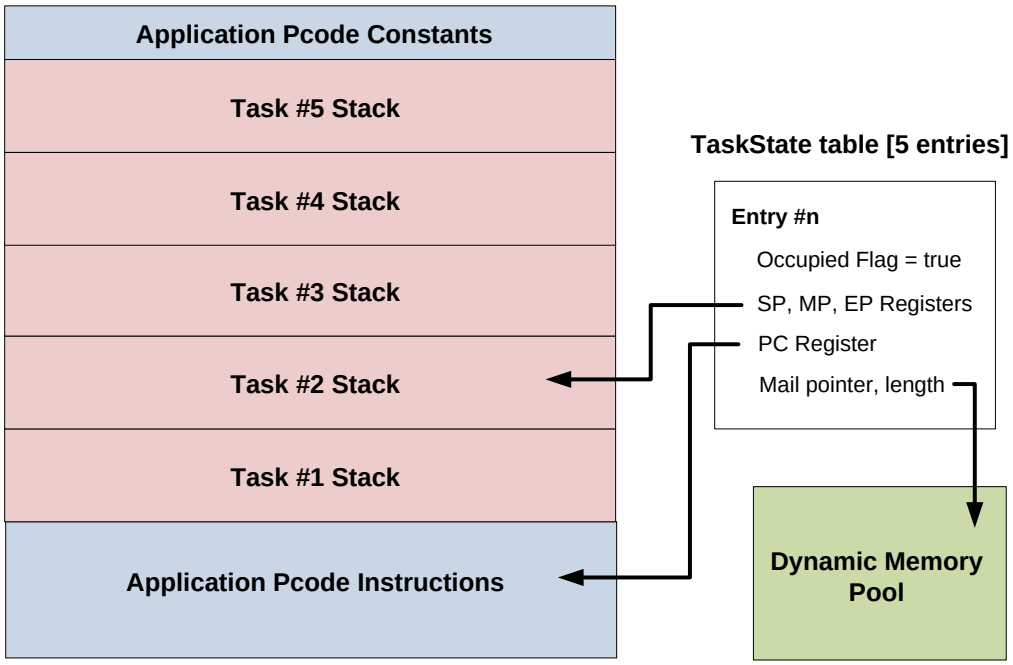

Figure 3-15 PMachine Modified for Concurrent Execution

The return value can then be used to cause each of the tasks to follow their own execution path depending on whether it is the parent or the child task. As shown in Figure 3-15, when a new task is created only four values need to be remembered to save and restore the task; namely the PC, SP, MP, and EP registers. PC is the program counter which is saved to remember what instruction this task was executing when it was suspended so that it can pick up where it left off when it starts running again. SP is the stack pointer which indicates where the current top of the stack is for this task. Since each new task has its own copy of the stack, each task must remember where that is in memory for its own stack. MP is the mark pointer which indicates where the base of the current stack frame is. EP is the extreme pointer which indicates the top of the current stack frame so that the VM knows where to put the next stack frame if another

procedure or function is called from the current context. There is no need to save any working registers because the PMachine is stack based and nothing else is stored in registers between instructions besides what is on the stack, making task switching relatively fast and efficient.

Another subfunction, TASKPEND, is to allow the currently executing task to voluntarily give up the virtual processor and allow a different task to run. The instruction searches through the TaskState table and finds another task that is ready and allows it to run by searching for the next higher task slot in the table until it finds one that is ready so that no task is starved. TASKWHO is called if a task needs to find out which task number it is. TASKKILL can be called for a task to permanently kill itself and allow other tasks to run instead. The slot then becomes available for a new task to claim via another call to TASKINIT.

The Pascal compiler was extended to support the new syntax and to output intermediate assembly code for it. The assembler was then extended to assemble the new extensions into a new bytecode instruction and stack-based parameters. Finally the interpreter was extended to support the new instruction and multitasking. The changes to support multitasking consisted mostly of adding the TaskState table to hold the registers that are saved, and four new functions, findEmptyTaskSlot(), saveState(), restoreState(), and TaskSwitch(). TaskSwitch() simply finds a new task to run, calls saveState() for the current task, and restoreState() for the new task.

5.7.1.4 Evaluating the Task Instruction

A Pascal program was written to exercise the new task instruction. The program creates five tasks with each task takes turns pending and allowing the next task to run for (10,000*5) iterations. The time to perform the actual task switching was calculated by taking the overall time and subtracting the time required to perform an empty loop with the same number of iterations. On the 12MHz target platform, the program took 17,660 ms to task switch 49,996

times which is 353 microseconds per task switch. The program stopped once the first task reached 10,000 iterations, which is the reason that it exited after 49,996 times rather than reaching 50,000 total iterations. Although this is an interesting calculation, which is used in later calculations, it is an intermediate result because the comparison to native code included mailing messages between tasks.

5.7.1.5 Implementing a Mail Instruction to Support Concurrency

A second instruction was then implemented to support message passing between tasks in the VM. The mail( ) instruction was created to handle this functionality:

```
Status := mail(function, parameter1, parameter2, parameter3);
 Function:
       MAILSEND=0
             parameter1: destination task ID
             parmeter2:  length of message to send
             parameter3: buffer address of message to send

       MAILRCV=1:
             parameter1: my task ID
             parameter2: size of message buffer for receiving
             parameter3: buffer address where to put received message
```

The MAILSEND subfunction sends a message from one task to another. It accepts an arbitrary message from the application which is normally an array or structure of data, obtains a dynamic buffer of the right size from the board support package (BSP) dynamic memory pool and copies the user's message into this buffer. This sequence frees the user's memory for other uses to avoid race conditions with it while the mail is being sent and used by the other task. When the mail message is sent, the address of the dynamic buffer is attached to the destination task in the TaskState table. The TaskState table was expanded to include variables to hold the address and length of the pending buffer. The message resides in the dynamic buffer, which is attached to the destination task until the task checks its "mailbox" to see that it has mail, which is accomplished via a call using the subfunction MAILRCV. In this implementation, a call by a

task to receive mail will cause the task to pend if no mail is available. The task then becomes ready to run again when mail arrives. The task that sent the mail continues to run until it voluntarily gives up the processor or pends on its own mailbox. This implementation was chosen because the native operating system used in the timing comparison performs this way allowing the timing comparison to be fair. Another faster implementation might be to allow the receiving task to wake up immediately which would make the system appear to switch tasks faster but is not commonly how embedded operating systems work.

The compiler was again extended to support the syntax for the new mail function along with its required parameters. The assembler was also extended to output the proper bytecode for the new instruction. Finally, the interpreter was extended to support the new mail messaging functionality by adding fields to the TaskState table and interfacing with the dynamic memory manager. The mail() operations were made to interface with many of the same routines used to support the task() operations since task switching and task pending can occur as an integral part of message passing.

5.7.1.6 Evaluating the Mail Instruction

A test program was then written to exercise the combination of the new mail instruction with the task instruction of the VM. In this program, 5 tasks are launched and each task creates a message ahead of time to send to the next task in sequence. It then suspends and waits to receive mail from the predecessor. Once it receives mail it then sends mail to the next task in sequence. To start the process, the original parent task sends a mail message to the next task. This sequence was repeated (10,000 * 5) iterations. Similar to the prior test application, empty loop timing was also performed yielding a net of 101,100 ms for 49,996 iterations (2.02 ms per iteration). As before, the program stopped once the first task reached 10,000 iterations, (thus

stoping before reaching 50,000 total iterations). This test was run without building the mail messages inside the loop to test just the tasking and mail without the supporting operations. A real application would need to build the mail message before each send operation, so the test was rerun with the building of the mail message inside the loop. This took a net of 239,630 ms for 49,996 iterations (4.79 ms per iteration). A measurement was also made of the entire loop which included processing the mail and the loop overhead. This measurement was made because the native execution includes the loop timing and the time required to build the message each time. This took 295,810 ms for 49,996 iterations (5.92 ms per iteration).

5.7.1.7 Performing the Same Operations in Native Code

The performance of the tasking and message passing in the VM was evaluated by comparing it to identical operations in a native environment. A part of proving the second hypothesis involves demonstrating that using high level abstraction closes the gap in performance between the native and VM environments. If there is no benefit to creating high level virtual instructions that perform major operations, then the VM would be shown to be ineffective which would disprove the hypothesis.

To be able to compare the virtual execution with native execution, the same mail test was run in a native execution environment. A multitasking environment was set up on the target platform using a Micrium µCos real-time operating system (Micrium, 2010). Five tasks were created along with mailboxes for each of them. Each task mailed a message to another task which unsuspended and then sent a message to the next task. It used the same dynamic memory manager that was used by the VM to buffer the messages. In this test 100,000 loops * 5 tasks = 500,000 iterations were performed in 400.2 seconds (800.4 ms per iteration).

5.7.1.8 Evaluating the Timing in Comparison with Native Execution

The ratio between the virtual execution and the native execution requires using the VM execution that included the full loop processing for a fair comparison. The ratio between virtual and native execution is (5.92ms / 0.800ms), or 7.40:1. Thus, the VM performed the same operations as the native operation just 7.40 times slower. As shown in Figure 3-16, this is a significant performance improvement compared to the 187:1 ratio for execution that was calculated for the standard mix of operations such as the Dhrystone test. The improvement is due more functionality combined into a single virtual instruction with both the task and mail instructions.

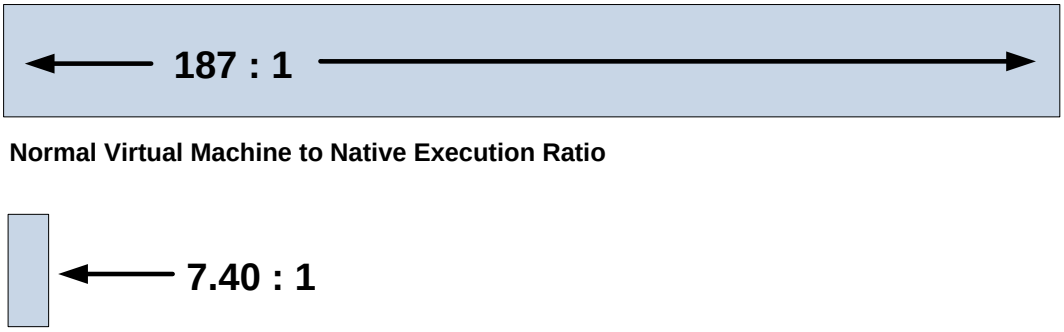

Figure 3-16 Comparison of VM and Native Performance with High Level Abstraction

5.7.1.9 Evaluating the Code Size in Comparison with Native Code

Another criterion for evaluating the effectiveness of the VM is code size. Both the native and virtual code from the research was measured and compared. First, the Dhrystone test was compared and then the tasking and mail test. The measurement for the tasking test included the code to declare and start the tasks, send and receive mail, and the outer loop to execute the task. As shown in Table 3-5, the native code was 44% smaller for the Dhrystone test, but the virtual code was 29% smaller for the tasking test. This result is explained because the Dhrystone test operates at a very low level of abstraction and the VM not only runs much slower but takes more

code space. The tasking test operates at a higher abstraction level and accomplishes more in a few powerful opcodes. It should be noted that this comparison does not include the 27K interpreter that is required to run the virtual application. The interpreter is a fixed size regardless of the size of the virtual application that it is executing.

Type	Code Size
Native Dhrystone Test	3,383 bytes
Virtual Dhrystone Test	6,091 bytes

Type	Code Size
Native Tasking Test	1,678 bytes
Virtual Tasking Test	1,188 bytes

Table 3-5 Virtual and Native Code Size Comparisons

5.8 Conclusion

A VM implementation was found that could be ported to a small microcontroller platform and made to run and support real-time performance. This result was achieved by porting and running a Pascal interpreter on an Atmel AVR microcontroller and then extending the language and interpreter to support high speed real-time serial communications. Timing results showed that the VM could maintain real-time performance even at the low end of the *small* microcontroller scale.

The VM was then extended with high level abstract language syntax and instructions and testing the performance. Multitasking and inter-task communications was added to the VM and the performance was tested using an application that spawned five tasks which passed mail messages repeatedly and timing the results. Test results showed that the performance ratio of the

VM compared to the native equivalent improved dramatically by increasing abstraction. Additionally, the code size for the virtual application was smaller than the native equivalent. This is consistent with some of the conclusions drawn in prior similar research within the field. Very early in the research Proebsting was able to show that the execution loop in an interpreter is very costly and the fewer bytecodes that needed to be executed the faster the program would run (Proebsting, 1995). Others showed that VMs are capable of doing things that native machine hardware is not effective at doing by implementing VMs that do not emulate real machines. The Esterel and the Harrison VMs are good examples of this (Plummer, Khajanchi, & Edwards, 2006) (Craig, 2005). Additionally, other research showed that abstraction is a way to increase performance in a VM as shown by the Carbayonia and Hume VMs (Gutierrez & Soler, 2008) (Hammond & Michaelson, 2003). Finally, it has been shown that creating a domain specific language is an effective way to create the "ultimate abstraction" for a given application (Hudak, 1996).

References

Abelson, H., Dybvig, R., Haynes, C., Rozas, G., Adams, N., Friedman, D., et al. (1998). Revised Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation* , 7-105.

Agosta, G., Reghizzi, S., & Svelto, G. (2006). Jelatine: A Virtual Machine for Small Embedded Systems. *JTRES '06 Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems* (pp. 170-177). Paris: JTRES '06.

Atmel Corporation. (2010, June 15). *AVR Solutions - 8 and 32 bit low power, high performance MCU*. Retrieved September 2, 2010, from Atmel: http://www.atmel.com/dyn/products/devices.asp?family_id=607#1965

Atmel Product Guide. (2011). *Key parameters for ATxmega128A1*. Retrieved April 13, 2011, from Atmel Corporation Product Guide: http://www.atmel.com/dyn/products/product_parameters.asp?category_id=163&family_id=607&subfamily_id=1965&part_id=4298&ListAllAttributes=1

Atmel XPlain. (2010). *AVR Xplain Series.* Retrieved November 16, 2010, from Atmel: http://www.atmel.com/products/AVR/xplain.asp?family_id=607

Brouwers, N., Corke, P., & Langendoen, K. (2009). Darjeeling, a feature-rich VM for the resource poor. *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys '09)* (pp. 18169-182). New York: ACM.

Brouwers, N., Langendoen, K., & Corke, P. (2009). Darjeeling, a feature-rich VM for the resource poor. *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys '09)* (pp. 169-182). New York: ACM.

Byte Craft Limited. (2010). *Catalog*. Retrieved December 19, 2010, from Byte Craft Limited: http://www.bytecraft.com/MPC_C_Compiler_for_Microchip_PIC

Craig, I. D. (2005). *Virtual Machines.* New York: Springer-Verlag.

Creasy, R. J. (1981). *The Origin of the VM/370 Time-sharing System.* Retrieved August 20, 2010, from http://pages.cs.wisc.edu/~stjones/proj/vm_reading/ibmrd2505M.pdf

Demichelis, A. (2010, August 12). *Squirrel the Programming Language*. Retrieved August 20, 2010, from Squirrel: http://squirrel-lang.org/

Dis Virtual Machine Specification. (1999, September 30). (Lucent Technologies) Retrieved August 20, 2010, from vita nuova: http://www.vitanuova.com/inferno/papers/dis.html

Eickhold, J., Fuhrmann, T., Saballus, B., Schlender, S., & Suchy, T. (2008). AmbiComp: A platform for distributed execution of Java programs on embedded systems. *AmI-Blocks'08, European Conference on Ambient Intelligence 2008.* Nurnberg.

Eller, H. (2005). *Optimizing Interpreters with Superinstructions.* Master's Thesis, Institut fur Computersprachen der Technischen Universit at Wien, Wien.

eLua - Embedded Lua. (2010, August 4). Retrieved August 20, 2010, from eLua Project: http://www.eluaproject.net/

GHI Electronics. (2012). *TinyCLR*. Retrieved March 20, 2012, from TinyCLR: http://www.tinyclr.com/

Gutierrez, D. A., & Soler, F. O. (2008). Applying Lightweight Flexible Virtual Machines to Extensible Embedded Systems. *Proceedings of the 1st workshop on Isolation and integration in embedded systems* (pp. 23-28). Glasgow: ACM.

Hammond, K., & Michaelson, G. (2003). Hume: A Domain-Specific Language for Real-Time Embedded Systems. *Proceedings of the 2nd international conference on Generative programming and component engineering* (pp. 37-56). Erfurt: Springer-Verlag.

Harbaum, T. (2006, June 13). *The NanoVM - Java for the AVR*. Retrieved April 21, 2011, from Harbaum: http://www.harbaum.org/till/nanovm/index.shtml

Hudak, P. (1996). Building domain-specific embedded languages. *ACM Computing Surveys (CSUR) , 28* (4es).

IAR Systems. (2010). *IAR Embedded Workbench for Atmel AVR.* Retrieved November 16, 2010, from IAR Systems: http://www.iar.com/website1/1.0.1.0/107/1/

ISO 7185:1990. (1990). Retrieved August 20, 2010, from International Organization for Standardization: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=13802

Kazi, I., Chen, H., Stanley, B., & Lilja, D. (2000). Techniques for Obtaining High Performance in Java Programs. *ACM Computing Surveys* , 213-240.

Levis, P., & Culler, D. (2002). Mate A tiny virtual machine for sensor networks. *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* , 85-95.

Lilith and Modula-2. (2010). (CFB Software) Retrieved August 20, 2010, from CFB Software: http://www.cfbsoftware.com/modula2/

Longbottom, R. (2010, November). *Dhrystone Benchmark Results On PCs*. Retrieved March 2, 2011, from Roy Longbottom's PC Benchmark Collection: http://www.roylongbottom.org.uk/dhrystone%20results.htm

Lua the Programming Language. (2010, July 29). Retrieved August 20, 2010, from Lua: http://www.lua.org/

Micrium. (2010). *uC/OS-II Kernel*. Retrieved December 20, 2010, from Micrium: http://www.micrium.com/page/products/rtos/os-ii

Microsoft .NET Micro Framework. (2010, August 19). Retrieved August 20, 2010, from Microsoft .NET Micro Framework: http://www.netmf.com/Home.aspx

Microsoft. (2011). *.NET Micro Framework.* Retrieved March 20, 2012, from Microsoft: http://www.microsoft.com/en-us/netmf/default.aspx

Microsoft. (2010). *Common Language Runtime Overview.* Retrieved October 21, 2010, from Microsoft Corporation: http://msdn.microsoft.com/en-us/library/ddk909ch

Microsoft. (2010). *Device Memory Management in the .NET Compact Framework*. Retrieved October 26, 2010, from Microsoft Corporation: http://msdn.microsoft.com/en-us/library/s6x0c3a4.aspx

Microsoft Visual Studio. (2005). *Microsoft Visual Studio 2005.* Retrieved May 30, 2011, from Microsoft: http://www.microsoft.com/visualstudio/en-us/products/2005-editions

Mono for Android. (2012). Retrieved March 20, 2012, from Xamarin: http://xamarin.com/monoforandroid

Moore, S. (2010, July). *http://www.standardpascal.org/p5.html*. Retrieved August 20, 2010, from Standard Pascal: http://www.standardpascal.org/p5.html

Novell. (2010, June 15). *Mono-project*. Retrieved August 20, 2010, from Mono: http://mono-project.com/Main_Page

Oracle. (2010, June). *Java Embedded FAQ*. Retrieved October 26, 2010, from Oracle Corporation: http://www.oracle.com/technetwork/java/embedded/overview/index.html

Oracle. (2008, August). *The Java Card 3 Platform.* Retrieved June 21, 2011, from Oracle: http://www.oracle.com/technetwork/articles/javase/javacard3-whitepaper-149761.pdf

Parrot. (2010, August 18). Retrieved August 20, 2010, from Parrot: http://www.parrot.org/

Plummer, B., Khajanchi, M., & Edwards, S. A. (2006). An Esterel Virtual Machine for Embedded Systems. *Proceedings of Synchronous Languages, Applications, and Programming (SLAP 2006)* .

Praus, F., & Kastner, W. (2008). *User Applications Development Using Embedded Java.* Retrieved April 21, 2011, from Praus: http://www.praus.at/files/knx08_embeddedjava_final.pdf

Proebsting, T. A. (1995). Optimizing an ANSI C interpreter with superoperators. *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (pp. 322-332). San Francisco: ACM.

Riedel, T., Arnold, A., & Decker, C. (2007). An OO Approach to Sensor Programming. *EWSN 2007 European conference on Wireless Sensor Networks.* Delft: EWSN 2007.

St-Amour, V., & Feeley, M. (2009). *PICOBIT: A Compact Scheme System for Microcontrollers.* Montreal: Université De Montréal.

STMicroelectronics. (2011). *Smartcard MCU with 32-bit ARM.* Retrieved June 21, 2011, from STMicroelectronics: http://www.st.com/stonline/stappl/st/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATA_BRIEF/CD00296560.pdf

STMicroelectronics. (2008). *ST23YL80 Smartcard MCU.* Retrieved June 21, 2011, from STMicroelectronics: www.st.com/stonline/books/pdf/docs/13982.pdf

Texas Instruments. (2010). *Microcontrollers (MCU).* Retrieved September 2, 2010, from Texas Instruments: http://focus.ti.com/mcu/docs/mcuprodoverview.tsp?sectionId=95&tabId=1531&familyId=916

The LLVM Compiler Infrastructure. (2010, April 27). (University of Illinois at Urbana-Champaign) Retrieved August 20, 2010, from LLVM: http://llvm.org/

Topley, K. (2002). *J2ME in a Nutshell.* Sebastopol: O'Reilly Media.

Virtanen, J. (2005). *GNU Pascal*. Retrieved August 20, 2010, from GNU Pascal: http://www.gnu-pascal.de/gpc/h-index.html

Weicker, R. P. (1988). Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules. *SIGPLAN Notices , 23* (8), 49-62.

Weicker, R. P. (1984). Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM , 27* (10).

Weiss, A. R. (2002, October 1). *Dhrystone Benchmark History, Analysis, Scores, and Recommendations.* Retrieved December 19, 2010, from EEMBC Certification Laboratories: http://www.eembc.org/techlit/datasheets/dhrystone_wp.pdf

Wirth, N. (1983). *Programming in Modula-2.* Springer-Verlag.

Wirth, N. (1970, October 30). *The Programming Language Pascal.* Retrieved August 20, 2010, from http://www-sst.informatik.tu-cottbus.de/~db/doc/People/Broy/Software-Pioneers/Wirth_hist.pdf

XDS Modula-2 Compiler. (n.d.). Retrieved February 22, 2011, from Excelsior LLC Web Site: http://www.excelsior-usa.com/xdsdl.html