Rappit: Framework for Synthesis of Host-Assisted Scripting Engines for Adaptive Embedded Systems

Jiwon Hahn, Qiang Xie, and Pai H. Chou
Center for Embedded Computer Systems, University of California, Irvine, USA
{jhahn, qxie, phchou}@uci.edu

ABSTRACT

Scripting is a powerful, high-level, cross-platform, dynamic, easy way of composing software modules as black boxes. Unfortunately, the high runtime overhead has prevented scripting from being widely adopted in embedded applications. This work proposes to overcome these obstacles by synthesizing light-weight, host-assisted scripting engines for embedded systems. The result is dramatically shortened development cycle due to the much higher-level abstraction, interactive access and dynamic reconfigurability, robust in-field software upgradability, and compact code size. This framework has been successfully applied to ultra low-power sensor nodes with under 10KB of program memory to high-performance platforms with fast Ethernet.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*General*; C.3 [**Computer Systems Organization**]: Special Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms

Design, Languages

Keywords

Scripting, adaptive systems, software synthesis

1. INTRODUCTION

Embedded systems have evolved from simple controllers to heterogeneous distributed systems that are a critical part of many key infrastructures today. To develop these systems, designers today write mostly C or assembly and can program the flash memory over a serial port in addition to burning EEPROM. Such development methodology has remained fundamentally unchanged for several decades. The code is often written at the lowest level of abstraction for a single processor with direct access to platform-specific registers such as timers, interrupts, and I/O ports. As a result, much development effort today goes into software – not so much for new applications, but for re-inventing these low-level abstractions for each intertwined combination of application and system functions. The single-processor, platform-specific approach is impeding progress while the rest of the world is moving towards larger scale, distributed networked embedded systems with increasingly dynamic configurations.

To address these problems with programming, we propose a new methodology that uses *scripting* as the primary way for higher-level software development and dynamic execution in either single-processor or distributed systems. Scripting has gained popularity in general-purpose computing, ranging from CGI-scripts, text processing, interactive animations, computer-aided design, and scientific computing. Ousterhout [13] has shown that scripting consistently achieves a 10× productivity gain over *system-programming* languages such as C or Java. Scripting is also a natural fit for loosely coupled systems.

What has prevented scripting from being widely adopted in embedded systems is the high runtime overhead. Most existing scripting engines are implemented for general purpose computers with virtually unlimited memory and high performance. There, convenience is the primary concern. For embedded systems, however, designers often try to optimize every bit of memory or speed to save component cost or increase battery life. As a result, they preclude the use of many scripting engines with full generality.

To solve this problem, we propose a methodology based on synthesis of host-assisted, light-weight scripting engines. Analogous to architecture description languages (ADLs) that target physical machines with the optimal amount of hardware resources for the given application, we synthesize a virtual machine with just the right complexity for the given application and architecture. Unlike Java VM, scripting can invoke powerful primitives interactively, without requiring recompilation. To make this lightweight, we take the *host-assisted* approach. That is, the end user gets the illusion that they are interacting with a nearly full-featured command-line prompt running on the embedded system, but much of the complexity is actually handled by the host. We limit the complexity by subsetting the language that the node needs to process.

The result is that designers are empowered with a much higher-level way of expressing the behavior of their systems while keeping the runtime overhead low. Not only can they design for yesterday's applications with much less effort, but they are now actually able to write much more powerful programs for a network of adaptive distributed embedded systems of tomorrow. Experiences have shown dramatically shortened development cycle due to the much higher-level abstraction, interactive access and dynamic reconfigurability, robust in-field software upgradability, and compact code size. This framework has been successfully applied to systems ranging from ultra low-power sensor nodes with under 10KB of program memory to high-performance platforms with fast Ethernet.

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

2. BACKGROUND AND RELATED WORK

Many embedded systems may be viewed as having a microcontroller (MCU) controlling I/O devices. The I/O may be for peripheral devices, communication with a host computer, or with another embedded system in a distributed network. We will use the application of wireless sensor nodes to illustrate the concepts, though the techniques are broadly applicable. Because these nodes should be small, low power, and low cost, most of them use small microcontrollers with very limited memory. For instance, some of the most widely used sensor platforms use the Atmel ATmega series of MCU with 1–4KB SRAM and 16–128KB program flash. Existing approaches can be divided into compiled vs. interpreted, and pre-deployment vs. post-deployment.

Many programmers take the compiled approach with little or no consideration for post-deployment update. That is, they write code for the bare machine because it incurs minimum runtime overhead, but it is very low level. Thin runtime support such as TinyOS [9] and BTnut [6] can be compiled in with the user program to make light-weight, monolithic executables. The problem is that they are rigid, and post-deployment firmware updates are difficult if not impossible, because it may be deeply buried inside a structure. TinyOS2 supports post-deployment writing of selective executable code into the program memory. However, in the case of heterogeneous networks, different binaries must be generated for each configuration. In all of these compiled approaches, pre-deployment testing is difficult. It would require either a simulator that does not have access to the actual I/O, or an in-circuit emulator (ICE) that may be difficult to set up if it has to be worn or mounted in a special way.

Scripting features have been considered to address limitations of compiled approaches. PyBAR [3] is a router that includes a python interpreter to facilitate introduction and maintenance of new services, though its performance is less than 1/10 of C++ implementation. Luxdbg [14], the LUxWORKS tool suite's debugger provides Tcl as a user-interaction language. Sensorware [4] is a framework designed for reprogramming the high-end sensor systems. The middleware occupies 237 KB, including a 64KB subset of Tcl called tinyTcl used mainly to support modular code update rather than interactivity. To further reduce the memory footprint, researchers propose a variety of interpreters. They interpret commands or virtual instructions that are encoded either in terms of the addresses of the routines to invoke (e.g., Forth [5]), or in terms of some arbitrary assignment of byte values (e.g., Java, Maté). Maté [10] is a bytecode interpreter built on top of TinyOS and occupies 16.8KB memory. Its bytecode is for low level operations for a stack machine, though they allow eight user-defined instructions. The support for scripting is thus very limited. Agilla [8] is a middleware system with a memory footprint of 41.6 KB of code and 3.59 KB of data memory. It is primarily for supporting mobile agents in a wireless sensor network, but not so much for system configuration or I/O control. In Pushpin [11], the application is composed of pieces of native code called *pfrags* (process fragments), which are dynamically transferred and executed on the Bertha OS inside each node. The Bertha OS occupies less than 14K, or 32K with nine pfrags.

Unlike other approaches that rely on fixed runtime support, we synthesize the scripting engine to explore a much wider range of implementation options, ranging from extremely lightweight to arbitrarily complex, depending on application requirements from above and resource constraints imposed by the architecture from below. Host-assist and script subsetting further push the limit of minimizing the footprint of the scripting engine. Table 1 compares the memory footprint of Rappit with the other techniques.

Name	Code Size	Data Size	Total
Sensorware	237 KB	$<$ 64 MB	$>$ 237 KB
Agilla	41.6 KB	3.59 KB	45.19 KB
Maté	16 KB	849 B	16.8 KB
Pushpin	32 KB	2.26 KB	34.26 KB
Rappit	1.4 – 16 KB	$<$ 1 KB	1.4 – 17 KB

Table 1: Memory footprints of runtime systems.

3. SCRIPTING APPROACH

Our approach to scripting is to blend the host and target systems (or "nodes") as one integrated environment. The nodes appear and can be accessed as data structures in the host's scripting environment. This section classifies subsets of scripting based on the runtime support needed on the node and illustrates their use with examples.

3.1 Language Classification

The scripting console on the host is a full-featured scripting interpreter. We base our syntax on the Python language [15] for convenience. By default, user commands to the nodes go through a preprocessor that has knowledge about each node. If there is no room or need for certain heavy-weight features, then they may be implemented on the host instead. We first classify scripting languages based on their interpretation requirements: script parsing, symbol table support, and memory management.

First, the designer must inform our synthesis tool of whether the node will require parsing in order to interpret the script. If the designer expects to type commands directly into the system without preprocessing (e.g., over a serial terminal), or if the commands are encoded in XML or one of the Internet application protocols (e.g., HTTP), then a parser needs to be generated. Because a great deal of the complexity may be in handling and recovering from syntax errors, the designer can further specify whether the parser needs to handle syntax errors or can expect all scripts to be well formed. If no script parsing is required, then the system can just run a bytecode interpreter. Orthogonally, if the script is expected to be transmitted over a lossy link, then CRC or additional data integrity attributes can be included.

The second issue is symbol tables. Even though scripting languages use symbols as logical addresses, symbol table support is not strictly required in many cases. At the lower level, a script accesses only symbols such as the names of the commands or hardware resources and a fixed set of symbols for software references. In this case, the host can translate all symbols into their numeric form without maintaining a symbol table on node. Script parsing may either use a symbol table, or the symbols may be hardwired. The latter may be more efficient when the scripting language is in an Internet protocol format, such as HTTP, because the parser only needs to extract a few relevant name-value pairs and ignore most other fields in the header.

Third, scripts have different requirements for memory. In the simplest case, the memory for variables and commands is statically allocated, besides the stack. Even if dynamic memory management is required, we push it to the host whenever possible if the host is part of the final runtime system. The host can assist with memory management by tracking the memory usage and send commands to move or copy memory segments.

3.2 Scripting Examples

Scripts can be typed by a human interactively in front of a terminal or by a GUI as the message interchange format. Scripts can also be executed in batch as high-level programs. We show exam-

ples on how scripting can be used for different purposes throughout the design and deployment stages.

3.2.1 Interactivity

Interactive access to the system at runtime is useful because it provides designers with instant control and observability to all relevant registers states, which are otherwise not observable without using scopes or debuggers via JTAG. Consider the following interactive session, where the user types into a text terminal connected to the embedded system (>> is the command-line prompt, and comments follow #):

```
>>  PORTA[1] = 1                # set port A pin 1
>>  PORTA[2] = !PORTA[2]        # toggle PORTA[2]
>>  PINA[0]                     # read input pin
0
>>  PORTA[2] = 1; PORTA[2] = 0  # toggle clock
>>  r5(PORTA[2]=1; PORTA[2]=0)  # toggle clock 5 times
```

3.2.2 System Configuration

One primary use of scripting is system configuration, at both design time and run time. During system design, it helps designers quickly experiment with the effects of different configurations without the lengthy, tedious edit-compile-load-reboot process. This feature supports in-field update for already installed embedded systems over any of the designated communication interfaces.

```
>>  mcu.sysclock = 1 MHz     # set system clock speed
>>  uart.baudrate = 9600 bps # set baudrate for UART
>>  rf.power = -5 db         # set RF tx power
>>  rf.speed = 1 Mbps        # set RF throughput
>>  rf.config                # query the configuration
{'payload': 1, 'power': -5, 'speed': 1000000, 'channel':
100, 'mode': 'TX'}
```

3.2.3 Multi-System Scripting

In addition to scripting an individual node, Rappit also supports scripting a set of heterogeneous nodes.

```
>>  L = rappit.listnodes()    # get list of visible nodes
>>  L
['node1', 'node2', 'node3', 'node4', 'node5']
>>  N = map(open, L)          # open connections to all
>>  dir(N[1])                 # see node N[1]'s keys & fcns
['ID', 'type', 'loc', 'adc', 'lcd', 'mcu', 'rf', ...]
>>  S = lambda x: x.every(50 ms, 'sample') # deferred eval
>>  map(S, N)            # issue command to all nodes
>>  N[4].stop('sample')  # tell node N[4] to stop sampling
```

Line 1 gets a list of nodes visible to the host. Line 4 calls open on each of the nodes and constructs a list of their proxy objects. The lambda expression on line 7 can be viewed as a script to be invoked at a later time. Line 8 tells each node to invoke its own sampling script once every 50 ms. Line 9 tells node N[4] to stop scheduling its sampling task.

4. THE RAPPIT FRAMEWORK

The Rappit framework supports the entire design flow from architecture modeling and synthesis to host-assisted execution of scripts. At design time, the user captures the target system architecture and behavior in a high-level description, whose syntax resembles the example scripts. To describe the architecture, the designer instantiates components from a component library, and makes the hardware interconnection either manually or automatically with an interface synthesis tool [7]. The next step is to select the desired subset of scripting features and commands, including the level of host assist allowed. The code synthesizer then composes the parser or bytecode interpreter along with all communication drivers and commands into an executable. The MCU is then loaded with this firmware, including some resident scripts.

The scripting engine can be synthesized and loaded early in the design cycle. This encourages designers to test features frequently and get instant feedback as they develop software. This is in contrast to compiled approaches, where designers tend to defer testing either to avoid the long compile/load cycles, or they must implement many features manually just to enable testing in the first place. For each node connected to the host, the host maintains a data structure for its current configuration so that the host can provide a translation service for the user console or GUI.

4.1 System Description

The Rappit framework provides libraries to aid description of system architecture, application, and communication. The component library captures the available features of hardware modules, including built-in devices on MCUs and stand-alone ones. An MCU model may include a system clock, on-chip memories, I/O interfaces (e.g., SPI), registers, I/O ports, and ADCs. The I/O ports are defined by their directions and widths. Stand-alone devices include off-chip memories or flash memory cards, RF tranceivers, sensors, LCD, joystick, speaker, etc. The below example shows the system description process.

```
# example: pin mapping for an RF module
import MCU                # load MCU module
mcu = MCU("ATmega169")    # instantiate an atmega169 MCU
import RF                 # load RF transceiver module
rf  = RF("nRF2401")       # instantiate a nRF2401 transceiver
rf.CS    = mcu.PORTB[0]   # connect the chip select pin
rf.CE    = mcu.PORTB[1]   # connect the chip enable pin
rf.DR1   = mcu.PORTB[2]   # connect the data ready pin
rf.CLK1  = mcu.PORTF[1]   # connect the clock pin
rf.DOUT1 = mcu.PORTF[2]   # connect the data pin
```

In addition to the drivers associated with the components, users may also add their own functions as well. An application is expected to be a script that invokes these modules already written or other scripts. Policies such as admission control, security, resource fairness, or energy saving are treated as directives already understood by existing software modules.

4.2 Scripting Engine Synthesis

The scripting engine to be synthesized depends on the message format between the host and the node. The messages can be either textual scripts or fixed-length bytecode packets. Synthesis of scanners and parsers for textual scripts is a well understood problem for either the host or the node. Here we focus the discussion on bytecode interpreters, which assume the host does the parsing and sends bytecodes. A bytecode interpreter is a switch statement in a loop and uses a stack for passing parameters. The difference between bytecode interpreters and parsers is mainly in how the parameters are handled. We define a fixed-length meta-format as a basis for encoding command and response packets. Synthesis directives also let the designer choose how many packets will be used for bytecode transfer, what kind of header and trailer bytes to add, and the CRC checking algorithm. For instance, if the communication link is reliable and there is only a single target system, then the `Dest` and `CRC` field can be omitted. Having a framework that maintains the format information enables the translation to be done consistently and correctly.

⟨*Command Packet*⟩

Dest.	Msg ID	Opcode	Arg1	Arg2	Arg3	CRC

⟨*Response Packet*⟩

Src.	Msg ID	Msg Type	Data Type	Data	CRC	EOP

Whether the interpreter parses scripts or runs bytecodes, they end up calling the same set of routines. The generated interpreter is compiled and linked with the communication driver, configuration

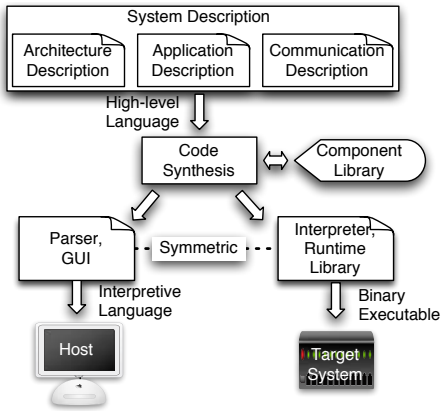


Figure 1: Code synthesis

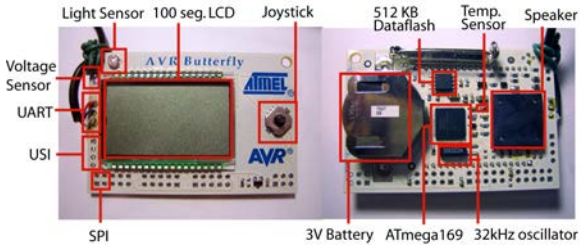


Figure 2: AVR Butterfly: an implementation platform

data, identification tags, and interrupt-driven job control to create the firmware for the target system.

4.3 Runtime Environment

The runtime environment spans the host and the nodes. The nodes may have different levels of complexity, and the host provides full generality and adapts its level of assist to what is needed by the node. The user interface on the host can be either a command line interface (CLI) or the Rappit GUI. The GUI provides an integrated, interactive control environment for selecting different communication links (e.g., UART, RF, USI) and for providing its own command prompt and the output window. Both the CLI and GUI generate textual scripts that are fed into the host parser. The host parser parses the script and links the lower level details retrieved from the runtime library. The host's packetizer encodes the parsed string into the command packet format and sends the packet to the target system. The target system runs a synthesized depacketizer to reconstruct the message. After dequeuing the message and before command execution, the admission controller decides whether to execute the command or not. The admission policy is defined by user, which can be resource-based, security-based, etc, and implemented at the time the firmware is synthesized. Upon admission, the interpreter starts executing the command as a subroutine call. The I/O drivers are the primitives inside the runtime library, which can access and manipulate the actual hardware.

5. EVALUATION AND ANALYSIS

The first prototype of Rappit has been implemented and tested on a number of MCUs, including the Atmel ATmega169, PIC16F series, FreeScale HC12 with fast Ethernet, and the nVLSI nRF24E1 with an 8051 core and 2.4GHz RF transceiver.

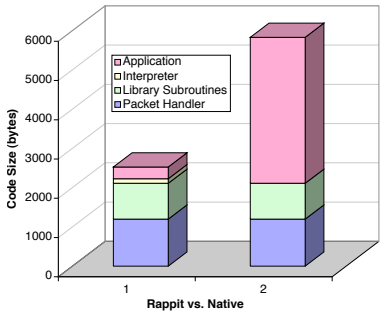


Figure 3: Code size: rappit vs. native code

5.1 AVR Butterfly

The AVR Butterfly board (Fig. 2) is the most feature-rich and the most resource-constrained. It has an Atmel ATmega169L 8-bit MCU at 8MHz, 512B EEPROM, 1KB SRAM, 16KB instruction memory, and peripherals including a dataflash, speaker, sensors (temperature, light, voltage), joystick, and an LCD. In our experimental setup, the command and response channel between the host and the board uses a USART serial link at 9600 baud. Both interactive and batch modes are supported. In interactive mode, packets are handled immediately in a nonblocking manner, whereas in batch mode, the entire script is loaded before execution.

5.1.1 Code Size

The total code size for the MCU includes the application, interpreter, library routines, and the packet handler. The interpreter and library routines are determined by code synthesis, which in turn depends heavily on user's directive, architecture constraints, and application needs. The packet handler implements the communication protocol, which is also synthesized. The application script is loaded at runtime and may go to either RAM or ROM. Our current implementation loads the bytecode into the 1KB RAM. In interactive mode, application storage can be a small (e.g., less than 10 bytes) buffer. We tested loading up to 280 commands in 3-byte packets of 255 different instruction types, for a total of 9.78KB of code size and 1KB of data size.

Fig. 3 compares the size of our scripting bytecode against compiled C. The test case contains 100 commands of 9 instruction types, which map to a mix of simple and complex primitives. The equivalent C code was written manually for the comparison. For cases where bytecodes map to powerful primitives, we can achieve dramatic reduction in application code size as shown in the top boxes in Fig. 3. The same subroutines and packet handlers were used for fair comparison. The interpreter itself consists a thin layer since it is a simple jump table.

5.1.2 Execution Delays of Rappit vs. Native Code

Execution delays are measured in terms of executed commands per second. We show results of two examples in Fig. 4, the two extreme cases (i.e., simple vs. complex) in our experiment. The first command sets and clears the register bits of the MCU. The second command samples the temperature for a certain amount of time, computes the average, and sends it to the host. Rappit is tested in both batch mode and interactive mode, where in batch mode all commands are preloaded to the RAM, while in interactive mode each command is sent through the serial link. In the result, the batch mode execution shows similar or better performance compared to the native code execution. Since the script size is smaller and can

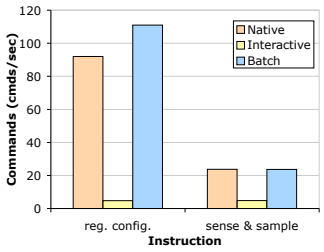
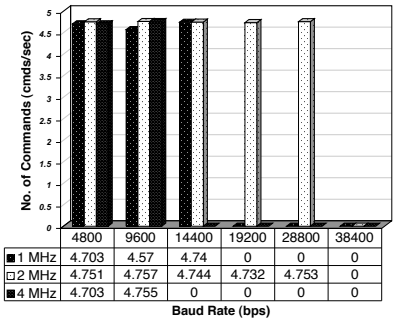
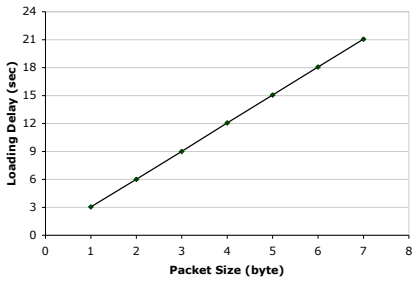


Figure 4: Comparison of execution delays



	4800	9600	14400	19200	28800	38400
1 MHz	4.703	4.57	4.74	0	0	0
2 MHz	4.751	4.757	4.744	4.732	4.753	0
4 MHz	4.703	4.755	0	0	0	0

Figure 5: Delay of loading commands vs. communication speed



Packet Size (byte)	1	2	3	4	5	6	7
Overhead (ms)	30.33	60.10	90.03	120.67	150.61	180.66	210.6

Figure 6: Loading overhead vs. packet size

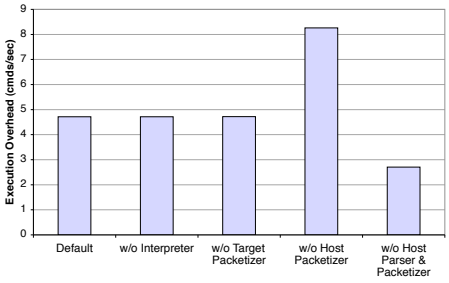


Figure 7: Execution overhead vs. different software components

fit entirely in RAM as opposed to loading from the much slower flash memory, the script in batch mode may run actually faster than the native code, although the speed-up is still limited by command decoding. In interactive mode, Rappit execution incurs a latency of 5 cmds/sec.

5.1.3 Overhead Components of Interactive Rappit

In interactive mode, the latency of 5 cmds/sec is tolerable, since users can continue typing ahead and sending commands while the results are being delivered. However, it is desirable to minimize the overhead. The overhead may be due to *communication*, *buffering* (i.e., queue, dequeue), *packets* (packetize, depacketize), and *interpretation*. The first three types of delay occur in loading the code to the board, while the last consists of the execution delays.

To reduce *communication overhead*, we explored different configurations of MCU's system clock and baud rate. The USART has certain combinations of parameters for the communication to be established and to provide a reliable link. Fig. 5 shows the valid communication configurations at different speeds. We observed that the total execution speed is independent of the link speed, and that communication is a negligible portion of the total overhead.

Buffering overhead is measured by comparing using the buffer against not using. Interestingly, the results show higher delay without buffering, since the buffer provides asynchronous writing, while without buffering, the read/write needs to be synchronized.

Packet overhead is measured by altering the packet size starting from a single `opcode` to adding fields in the order of `arg1`, `msgID`, `dest`, `arg2`, `arg3`, and `crc`. Fig. 6 shows the result of loading 100 commands for testing each packet size. The results show that the packet overhead becomes a major bottleneck in the performance for longer packet sizes, especially if CRC check has to be done in software.

Interpretation overhead is practically a negligible part in the execution time. The execution time shows little difference for an interpreter with a range of numbers of instructions from zero to 255.

In addition, we generated several test cases to analyze the overhead. Fig. 7 shows disabling the interpreter and the packetizer in the target system makes little difference. However, the host side shows more influence on the overhead, where the parser compresses the raw string, but in turn, the packetizer adds headers and trailers. The results confirm that our methodology adds some overhead not so much to the core execution, but mainly on the communication side, which can be improved by the adjustment of protocol and data format.

5.2 The Mini-FDPM Breast Cancer Detector

We have successfully applied the Rappit methodology to the development of the Mini-FDPM system [12]. It performs broad-

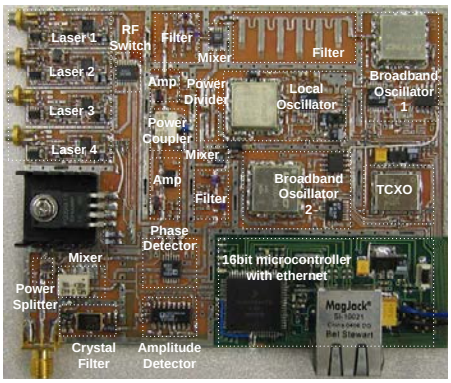


Figure 8: The Mini-FDPM System

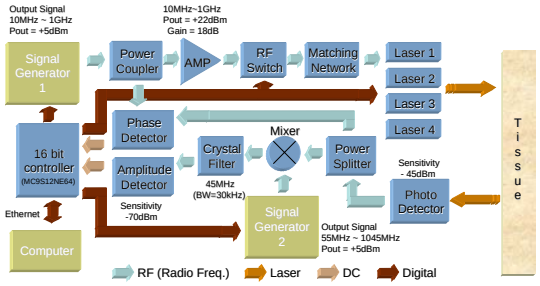


Figure 9: System Block Diagram of the Mini-FDPM

band modulation on the intensity of near-infrared laser diodes and derives the scattering and absorption coefficients of the bulk tissue from phase and amplitude data measurements. It consists of a broadband generator, laser modulator, and detector as shown in Fig. 9. It also includes the FreeScale MC9S12NE64 16-bit MCU with an integrated 10/100 Mbps Ethernet controller.

The MCU controls the peripheral devices including the frequency synthesizers and the laser drivers through I2C, SPI, and general-purpose I/O (GPIO) pins, and performs measurements with its built-in ADC. The samples are sent back to the host computer through the Ethernet interface. The MCU runs a command interpreter with routines for accessing the built-in Ethernet, SPI, I2C, ADC, and GPIO. The total code size of the interpreter is only 10.2KB. The implemented system is shown in Fig. 8. The Rappit GUI opens a terminal in the host to control the MCU through the Ethernet interface. To collect measurement data, developers can simply type the commands or invoke the GUI to generate the following script:

```
>> sc              # configure SPI interface
>> PH[3] = o       # set port H[3] to output
>> PH[3] = 1       # turn on the laser
>> i2s197          # send data to I2C device 2
>> i1s197          # send data to I2C device 1
>> ss98            # send to SPI device(control frequency)
>> r5(a2;a4;a6)    # read ADC channel(2, 4, 6) 5 times
>> save result     # save data to file 'result'
```

It takes the developer only a few minutes to write the script, and the result can be collected and post-processed (e.g., FFT) on the host immediately. This enables the user to easily customize the behavior of the instrument dynamically over different applications, without low-level programming. As we revise the Mini-FDPM designs with a different, newer MCU, the same script and the rest of the software will continue to work correctly.

6. CONCLUSION

Embedded systems are long overdue for an extreme makeover in software. Traditional compiled languages for uniprocessors are no longer keeping up with the trend towards distributed, loosely connected embedded systems. Scripting represents one of the most promising solution and has demonstrated compelling advantages in general-purpose computing. The interactive nature not only shortens development cycle and encourages testing but also makes these systems adaptive by design. What is sorely needed is to overcome high-overhead challenges that have prevented scripting from wider use in embedded systems. The Rappit framework described in this paper represents a first step towards bringing more viability of scripting to highly constrained embedded systems. Analogous to synthesizing a processor from an ADL (architecture description language) by allocating the right amount of resources to better fit the workload of an application, we synthesize a software machine, namely the scripting engine that best fits the architecture below and applications above. We exploit host assist and synthesis to make this technique applicable to a wide range of architectures, from very small MCUs that rely heavily on host assist, to high-performance systems that implement fully general scripting. Results show that scripting as synthesized by our Rappit framework is not only viable and convenient but actually has code size advantages over native code while incurring minimal overhead, even on one of the most resource-constrained MCUs.

Acknowledgments

This work was supported by the National Science Foundation grant CCR-0205712 and an NSF CAREER Award CNS-0448668, and sponsored in part by the National Institute of Health through an NTROI (Network for Translational Research on Optical Imaging) seed grant. The authors thank Chan Woong Nam from LG Electronics for the careful feedback and help in performing the experiments.

7. REFERENCES

[1] Atmel Corporation. In *http://www.atmel.com/*.

[2] J. Axelson. Tiny and inexpensive programmable controllers for quick project development. *MicroComputer Journal*, pages 20–27, May 1995.

[3] F. Baumgartner, T. Braun, and B. K. Bhargava. Design and implementation of a python-based active network platform for network management and control. In *IWAN '02: Proceedings of the IFIP-TC6 4th International Working Conference on Active Networks*, pages 177–190, London, UK, 2002. Springer-Verlag.

[4] A. Boulis, C.-C. Han, and M. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobySys*, 2003.

[5] L. Brodie. *Thinking Forth*. Prentice Hall, 1984.

[6] BTnut System Software. In *http://www.btnode.ethz.ch/support/btnut_api/main.html*.

[7] P. Chou, R. B. Ortega, and G. Borriello. Interface co-synthesis techniques for embedded systems. In *Proc. of 1995 International Conference on Computer-aided Design*, pages 280–287, November 1995.

[8] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless wireless sensor network applications. In *24th Internatonal Conference on Distributed Computing Systems (ICDCS'05)*, 2005.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.*, 34(5):93–104, 2000.

[10] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[11] J. Lifton, D. Seetharam, M. Broxton, and J. A. Paradiso. Pushpin computing system overview: A platform for distributed, embedded, ubiquitous sensor networks. In *Pervasive '02: Proceedings of the First International Conference on Pervasive Computing*, pages 139–151, London, UK, 2002. Springer-Verlag.

[12] K. S. No and P. H. Chou. Mini-fdpm: a handheld non-invasive breast cancer detector based on frequency domain photon migration. In *Proc. of IEEE BioCAS*, pages 2.2–5–2.2–8, December 2004.

[13] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[14] D. E. Parson, B. Schlieder, and P. Beatty. Extension language automation of embedded system debugging. *Kluwer Academic Publishers*, 9:7–39, January 2002.

[15] G. van Rossum. Extending and embedding the python interpreter. *Amsterdam: Stichting Mathematisch Centrum*, 1995.