TinyVM: An Energy-Efficient Execution Infrastructure for Sensor Networks

Technical Report CS-TR-9003
Department of Computer Science
Yonsei University, Seoul, Korea

Kirak Hong‡, Jiin Park‡, Taekhoon Kim‡, Sungho Kim‡, Hwangho Kim‡, Yousun Ko‡, Jongtae Park‡, Bernd Burgstaller‡ and Bernhard Scholz*
‡Yonsei University, Seoul, Korea *The University of Sydney, NSW 2006, Australia
bburg@cs.yonsei.ac.kr scholz@it.usyd.edu.au

ABSTRACT

Energy-efficient implementation techniques for virtual machines (VMs) have received little attention yet: conventional wisdom claims that VMs have a diametrical effect on energy consumption and sensor network applications on VMs are therefore short-lived.

In this paper we argue that bytecode interpretation is affordable for sensor networks if we synthesize VMs specifically for energy-efficiency. We present TinyVM, an execution infrastructure that seamlessly integrates with nesC/TinyOS and C-based environments. TinyVM achieves high code density through the use of compressed bytecode as the primary program representation. Compressed bytecode allows rapid application deployment with low communication overhead. TinyVM executes compressed byte-code in-place, which eliminates the need for a decompression stage and thereby reduces memory consumption on sensor nodes.

Our infrastructure automates the creation of energy-efficient application-specific VMs. Applications are partitioned in machine-code, byte-code and VM instruction set extensions. Partitioning is manually controlled and/or fully guided by a discrete optimization problem that produces a partitioning with lowest energy consumption for a given memory space constraint. We provide experimental results for sensor network benchmarks and selected applications on Atmega128-based motes and the Intel iMote2.

1. INTRODUCTION

A VM is a software execution platform that abstracts from native execution on a processor. VMs for wireless sensor networks have received a lot of attention in the recent past [29, 25, 34, 23, 18, 1, 22] to overcome the difficulties of programming sensor networks at massive scale for a range of platforms. Various hardware platforms for sensor nodes [32, 15, 16, 3, 30] make software development difficult, although the actual patterns are uniform across platforms. An execution infrastructure that builds on VM technology gives the ability to build heterogeneous systems, allowing cheap dynamic code updates, a uniform programming interface, and high code density for resource constraint sensor nodes. Higher code density has substantial advantages for WSN applications: (1) more code can be packed in small memories, (2) smaller code sizes enable larger local data caches which may reduce network traffic, (3) smaller code sizes enable in-situ deployment over the radio because of the reduced transmission costs and energy, and (4) due to reduced memory footprint, programs are faster and more energy-efficiently written to flash memory.

Sensor networks impose significant challenges for VM-based execution infrastructures. Hardware platforms range from micro-controllers [16] to 32bit RISC-based platforms [30]. A VM technology is required to cope with the resource limitations of micro-controllers while it must be able to run large applications on a 32bit RISC platform with far more resources. Energy-efficiency is another challenge: the execution speed of VMs is slower than native execution on a processor and hence will consume more energy than machine code execution. Ertl et al. [10] estimate that VMs impose a slowdown between a factor of 10 and 1000, depending on the implementation technique. Current VMs for wireless sensor networks do not incorporate modern VM techniques such as just-in-time decompression of compressed bytecode and removal of interpreter dispatch loops [20] and impose a slow-down of an order of magnitude of two to three. For wireless sensor networks, C and nesC/tinyOS [12] are the de-facto standards for programming sensor nodes. An execution infrastructure that weaves into nesC and C is integrated easily with existing wireless sensor network projects whereas VMs introduced in the past do not have the capabilities of running system code nor have the robustness to run generic C code. VMs for sensor networks require mixed-mode execution: efficiency considerations and space constraints suggest a partitioning of a WSN application into a part that is executed as bytecode, and a part that is executed as machine code. Mixed-mode execution requires an efficient bi-directional calling-mechanism between machine code and bytecode. Bytecode calling machine code is common, e.g., with application-specific VMs (ASVMs), but the opposite direction is less obvious: it arises, e.g., when an operating system scheduler (machine code) starts the execution of a bytecode task. Other examples include machine coded device drivers performing up-calls to bytecode functions.

The contributions of this paper are as follows.

- We present an execution infrastructure that, for the first time, enables the compilation of nesC (and C) applications to an energy-efficient combination of compressed bytecode, machine code, and application specific instructions. This partitioning is either performed manually and/or automated.
- We introduce a fast just-in-time decompression decoder that is able to execute Huffman-compressed instructions using different Huffman codes for instructions and operands.
- We model the energy/space trade-off between machine and bytecode execution as a discrete optimization problem. This optimization problem is parameterizable with respect to optimization objectives such as execution energy and code size. This allows domain experts to safely explore the design space of the system: based on the provided optimization objectives, our programming environment automatically performs an optimal partitioning.
- We conduct extensive experiments on Mica2 and iMote2 to demonstrate the validity of our approach.

The remainder of this paper is organized as follows: in Sec. 2 we survey related work. In Sec. 3 we introduce the overview of our TinyVM execution infrastructure. In Sec. 4 we describe the techniques used to implement the VM. Sec. 5 is devoted to the machine/bytecode partitioning of an application, and Sec. 6 presents our evaluation of TinyVM on Mica2 and the Intel iMote2. We draw conclusions and outline future work in Sec. 7.

2. RELATED WORK

Sun Microsystems introduced the SunSPOT platform for developing WSN applications in Java [34]. Their VM supports J2ME and runs on an ARM platform using an of the orders of magnitudes more power than an 8-bit microcontroller platform. Sentilla Corp. introduced a VM of the same name with fewer resource requirements. In [25] another VM for Java has been introduced that implements a subset of Java. SwissQM [29] is a VM targeted at query processing in WSNs. The SwissQM bytecode is a subset of Java VM bytecode plus special-purpose instructions for query processing and sensor access. SwissQM is oriented for data-acquisition. User-defined functions are compiled to bytecode, to be used in queries. Although the bytecode instruction set is Turing-complete, SwissQM supports only a single integer data type, no arrays and no function call mechanism. VMStar [18] is a synthesizable scalable platform for WSNs. It is based on Java, offers application-specific VM synthesis and dynamic software updates. The VMStar runtime environment provides a native interface for device-specific functions, and support for dynamic memory management and scheduling. Like TinyVM, VMStar employs threaded code. VMStar performs Java Class file compaction but no bytecode compression.

Maté [22] is a communication-centric VM implemented on top of TinyOS. Maté stores bytecode in so-called capsules that are 128 byte in size. Every bytecode instruction is one byte long. The abstraction level of the Maté instruction set is very high, resulting in small program sizes. Maté is directly programmed in bytecode; it supports instruction set extensions through 8 user-definable instructions. One of Maté's key goals is WSN reprogrammability; code capsules are injected into the network and distributed through a viral code distribution scheme.

Application-specific VMs (ASVMs) are a recent extension of Maté [23] that trades portability for performance. Portions of code are encapsulated in application specific extensions of the VM. This results in very high code density, low propagation cost and low interpretation overhead. We adopted the idea of ASVMs for TinyVM: in TinyVM, users can extend the VM's instruction set through machine code functions or through instruction set extensions for the interpreter itself. Both extension mechanisms are transparent to the TinyVM nesC-to-bytecode translation mechanism.

DVM [1] is a VM for multiple applications and reprogramming of WSNs on top of SOS. The DVM instruction set can be extended at runtime.

TinyVM differs from the above approaches in the following: (1) it uses in-place execution of compressed bytecode to reduce the memory footprint of programs, (2) it employs discrete optimization to address the trade-off between memory footprint and energy consumption, and (3) it is able to execute nesC on TinyOS, (4) allows call-sites in native machine code to execute seamlessly functions represented in compressed bytecode[1], and (5) it is a general purpose VM that is capable of executing the C programs of the SPEC2K [35] benchmark suite whereas other VMs and their languages for WSN are severely restricted in their expressiveness. A prior version of this work appeared as a poster abstract [17].

In [4] a virtual machine was designed for C that uses off-line compression, i.e., compressed bytecode is expanded in a memory buffer and converted to an expensive internal VM representation called threaded code. Threaded code ensures fast execution but consumes four times as much memory as machine code [4, Table 3]. The memory buffer further increases memory consumption. This virtual machine is not viable for micro-controller based WSN platforms due to the high memory consumption.

Compression of bytecode has been proposed by [20]. It allows to execute bytecode in its compressed form (see Fig. 6(b)). The fetch mechanism of the VM is extended to decompress bytecode on-the-fly when it is executed. This technique overcomes the disadvantage of a traditional approach by making the memory buffer for execution redundant. Our method differs from the method introduced in [20] by splitting the instruction stream into opcodes, literals and addresses and encode these streams separately with different Huffman codes. This method results in high compression rates.

The implementation of VMs have been significantly alleviated in the presence of high-level virtual machine generators such as VMgen [10], which takes VM instruction descriptions as input and generates C code for the VM, disassembler, tracer and profiler. However, VMgen produces threaded code, which is fast but its internal representation consumes four times more memory than machine code.

TinyVM is a node-centric programming model in the sense that the programmer thinks in terms of single (sensor) nodes operating in the context of the WSN application. This contrasts data-centric approaches like TinyDB [26] or the role assignment programming paradigm [33]. TinyVM facilitates high-level WSN programming approaches.

3. OVERVIEW

We designed TinyVM for high performance and efficient mixed-mode execution capabilities. The execution environment of TinyVM consists of the VM and the application image as depicted in Fig. 1. The application images contains compressed bytecode and machine code. Machine code executes on the mote's CPU core, whereas bytecode executes on the VM. In TinyVM the thread of execution can jump back and forth between the machine and the bytecode realm seamlessly, which we refer to as *mixed-mode execution*.

[1]This feature is important to enable arbitrary partitioning of the code between byte code and machine code. If not available, leaves in the static call tree (or subgraphs with no outgoing calls) only can be represented in machine code.

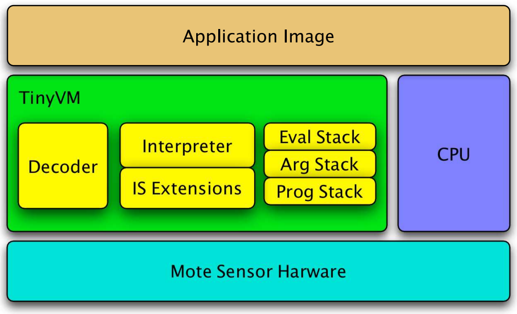


Figure 1: TinyVM Execution Environment.

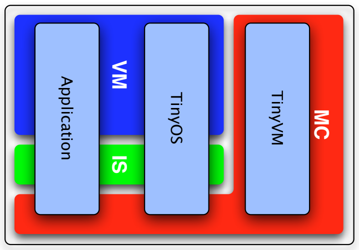


Figure 2: Partitioning into Machine-Code(MC), Instruction Set Extensions(IS), and Bytecode(VM).

As outlined in Fig. 2 the TinyVM execution infrastructure allows a partitioning of code into three portions: machine code (MC), instruction set (IS) extensions, and bytecode (VM). The partitioning is not bound to the application itself and may span across the operating system since TinyVM runs on the bare metal.

The partitioning into machine-code, application specific instruction set extension, and bytecode requires a mechanism to map source code to machine code, VM extensions, and bytecode. In TinyVM we have the partitioning automated and/or the programmer provides annotation in the source code or in configuration files. The automated partitioning seeks for minimal energy consumption under a given space constraint (see Sec. 5) and can partition the code into compressed bytecode and machine code. The annotations of the programmer make explicit which portion of the code is executed on the VM, on the CPU and as an application specific instruction set extension, respectively. TinyVM introduces three optional code qualifiers, i.e., `__vm__`, `__mc__`, and `__is__`, to assign nesC functions, commands, events and tasks to the domains. In absence of code qualifiers, the translation system selects between compressed bytecode and machinecode. Examples for the annotation approach are given below:

```
void __vm__ task do_all() { ... }
void __mc__ f_do_all()    { ... }
int __is__ log2(int x)    { ... }
```

In this example function `f_do_all` is compiled to machine code, task `do_all` is compiled to compressed bytecode, and function `log2` is synthesized as an application specific instruction set extension. To be able to synthesize function `log2` as an application specific instruction extension, function `log2` must not call a function that is represented in bytecode.

An overview of our compilation framework is shown in Fig. 3. The inputs to our framework include the user-supplied nesC sources, the configuration and profiling information ("config and profile info"), the interface of the TinyVM nesC component, and the specification of the VM ("TinyVM spec"). NesC source files, configuration information and the TinyVM nesC interface are input to the nesc compiler/splitter ("ncc & splitter"). The splitter is a modified ncc compiler that translates nesC sources to C, thereby partitioning them into two files: one C file containing the code to be compiled to bytecode, and one with the code to be compiled to machine code. Splitting is performed according to the given configuration information, code qualifiers, and program optimization. C code that is to be compiled to machine code is subjected to the GCC compiler to translate it into an object file ("machine code"). C code that is to be translated to bytecode is submitted to the bytecode backend of LCC. The resulting bytecode is then put through the optimizer ("VMOPT"). The optimizer uses the user-defined interpreter extension specification (see Sec. 4.1) to patch those procedure calls, for which a user-defined VM instruction exists. The optimizer produces a list of VM instructions that occur in the optimized bytecode. This list is used during VM synthesis to decide on the VM instructionset. The optimized bytecode is run through our Huffman-encoder to produce a bit-stream blob of encoded bytecode ("bytecode image"). The Huffman-encoder generates the decoding tables for the opcode-, literal-, and address bitstream; these tables belong to the VM ("TinyVM image"), together with the synthesized VM instruction set and the interpretive function. For the generation of TinyVM we use VMgen [10] that is a high-level code generator for VMs. We have rewritten the backend of VMgen to execute compressed bytecode[2]. The GNU linker ("LD") links machine code, bytecode and VM into an application image to be deployed on the mote hardware.

4. TINYVM RUNTIME ENVIRONMENT

TinyVM is a stack-based VM. Its basic architecture is depicted in Fig. 1. Mixed-mode execution requires a seamless transition between compressed bytecode and machine code. For this purpose special trampolines need to be injected before the bytecode of a function, that allow the invocation of the VM if a bytecode function is called from machine code. The VM needs to have a dedicated call mechanism for machine code functions as well [4].

TinyVM consists of the VM instruction decoder, the VM interpreter, and the VM evaluation- and procedure call argument stacks (VM Eval Stack and VM Arg Stack in Fig. 1). "Prog Stack" is the stack used for the execution of machine code. The core instruction set of our VM is related to the bytecode interface that comes with LCC [14], with the main deviations being induced by the requirements of the seamless integration of bytecode (`__vm__`) and machine (`__mc__`) code execution and by application specific virtual machine extensions for WSN application. Our VM has only few hardware-dependent parts, i.e., (1) the 8 bytes of assembly code employed with trampolines, and (2) the target procedure calling conventions that we need for mixed-mode execution. TinyVM does not use any system calls, which makes it able to run on the "bare" CPU. We synthesize the instruction set according to the needs of the respective application, to keep the memory footprint of the VM low. TinyVM is available on IA32 under Linux and Windows, ARM-based architectures such as the Intel iMote2 and on Atmega128-based motes such as Mica, Mica2, MicaZ and BTnodes.

4.1 Instruction Set Extensions

ASVMs [23] allow the user to define the abstraction-level of the VM instruction set: on the one extreme, an instruction set may consist of just one instruction with the semantics of "execute the program". On the other extreme, we might have a multitude of instructions which perform only insignificant amounts of computation. A high-level instruction set clearly increases the density of the bytecode, but it can severely increase the footprint of the VM's instruction set implementation. It is a technique to partition the code in machine code and byte code whereas machine code is encapsulated in new instructions of the virtual machine. Extending the instruction set of a virtual machine and removing existing instructions cannot be done on the fly for virtual machines. Hence, re-occurring patterns/code fragments including sending and receiving of data are synthesized as new instructions of the Virtual Machine. In contrast to mixed-byte code execution where machine and byte-code can be mixed arbitrarily without having any impact on the virtual machine. Our execution environment gives the designer the choice whether to pack code fragments into instructions or whether instructions

[2]VMgen provides the execution of threaded code only; threaded code is an intermediate representation that consumes four times more memory than machine code.

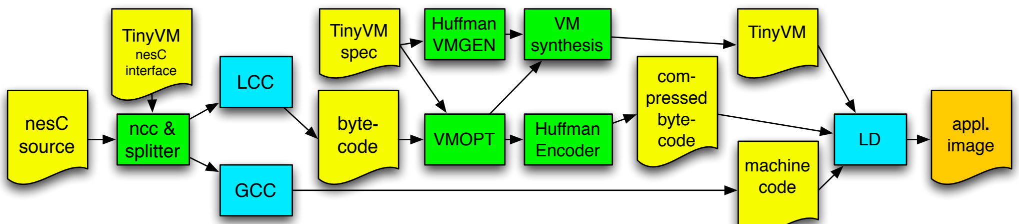


Figure 3: TinyVM Compilation Path.

are executed in a mixed-mode execution environment. The later has more overhead when it comes to switch between the VM and the machine code execution. The former has the advantage of being faster but there is less flexibility as soon as the VM is uploaded to a sensor node. With TinyVM, we provide two mechanisms to extend the core instruction set by user-defined instructions.

Loose Extensions: compiling a function to machine code already represents an extension of the VM by a highly-efficient implementation of a coherent piece of source code. This extension is *loose* in the sense that the new instruction is executed via a procedure call. The advantage of loose extensions is their ease of use: implementing a coherent piece of source code as a C/nesC function and assigning it to be compiled to machine code is all that is requested from the TinyVM user. Dynamic code updates can be applied with loose extensions. Techniques for such dynamic code updates were introduced, e.g., in [28, 8], they are however outside the scope of this paper. The disadvantage of loose extensions is their loose coupling with the VM: executing the new instruction comes at the additional cost induced by the procedure call. Loose extensions are entirely covered by TinyVM's mixed-mode execution capabilities.

Interpreter Extensions: extending the VM's interpreter with instructions avoids the procedure call overhead associated with loose extensions. Interpreter extensions execute as efficient as instructions from the core instruction set. In the remainder of this section we develop our interpreter extension mechanism in the context of VMgen instruction set specifications.

As an example for an interpreter extension, we consider the `merge` instruction of SwissQM[29]. SwissQM provides the `merge` instruction to perform in-network aggregation along the path to the WSN gateway. This instruction has the form

$$\texttt{merge}(\mathrm{n}, \mathrm{m}, \mathrm{aggop}_1, \ldots, \mathrm{aggop}_n),$$

where n, m, and aggop_{1-n} are parameters of the merge operation. Suppose that `merge` is a coherent and self-contained computation that the user wants to add to the TinyVM instruction set. The envisioned instruction set extension consists of two steps.

In the first step, we provide a C-level declaration for the new instruction, `void merge(int n, int m, ...);` To be accessed from nesC, this declaration can be wrapped into a nesC interface, similar to [23]. Calling the operation `merge(n,m,op1,op2);` from C/nesC will result in bytecode depicted in Fig. 4. Therein the arguments are pushed in reverse order onto the VM argument stack. In line 15 the number of arguments is pushed onto the VM evaluation stack. Thereafter the call to `merge` is performed.

```
1  addrl_p4 12
2  indir_i4
3  arg_i4       #push op2 on Arg stack

4  addrl_p4 8
5  indir_i4
5  arg_i4       # push op1 on Arg stack

9  addrl_p4 4
10 indir_i4
11 arg_i4       # push m on Arg stack

12 addrl_p4 0
13 indir_i4
14 arg_i4       # push n on Arg stack

15 cnst_i4 4    # push number of args
16 call merge   # procedure call
```

Figure 4: Bytecode to call the `merge` operation.

We use a post-processing step that replaces the call instruction in line 16 by the user-extended instruction `merge_v`. This post-processing step takes place before the bytecode is encoded. In Fig. 3 the postprocessor is labeled as "VMOPT".

In the second step, the TinyVM user has to provide a VMgen instruction specification for the `merge_v` instruction. Due to the nature of the TinyVM call-mechanism user-defined instructions get their arguments form the Arg stack instead of the evaluation stack. Apart from that, the specification works in the same way as core instruction specifications. To conclude this example, we give a pseudo-code specification for the `merge_v` instruction in Fig. 5. VMgen provides the ARG prefix to denote arguments form the VM argument stack. In this way the user-provided instruction specification can reference instruction arguments via `ARGn` and `ARGm` (see e.g., line 6 in Fig. 5). The evaluation stack is the default stack, which means that we do not need to prefix the argument count `argc`. Note that `argc` denotes the

```
1 merge_v (argc ARGn ARGm -- )
2 for(int i=0;i<argc;i++) {    # process args
3       foo(ARGn, argc);       # do something
4       ...
n }
```

Figure 5: Instruction Specification for `merge_v`.

number of arguments (see Line 15 of Fig. 4). The actual implementation for `merge_v` will consist of the C code provided by the user from Line 2 of Fig. 5 onwards.

Our interpreter extension mechanism facilitates the extension of the core instruction set with arbitrary VM instructions without modifying the bytecode compiler backend. Modifications of the compiler backend would make interpreter instruction set extensions impractical. Instead, we integrated our instruction set extension mechanism with VMgen's instruction set specifications. We believe that this mechanism makes user-supplied interpreter extensions applicable even for domain-experts who might not be too fluent in interpreter technology. It should be noted that dynamic code updates cannot be used with interpreter extensions, because the instruction extension is part of the VM's interpreter function itself, and functions cannot be partially updated (we would have to replace the whole interpreter). The advantage of interpreter extensions vs. loose extensions is that (1) an interpreter instruction avoids the procedure call overhead, and that (2) VMgen can perform peephole-optimizations across instructions. These optimizations are e.g. necessary to use superoperations (sequences of VM instructions) to further increase the performance of the VM [10].

4.2 Execution of Compressed Bytecode

Memory is at a premium in wireless sensor nodes, and code compression is a key technique to accommodate complex applications in small memory sizes. Code compression is an established technique [9, 11, 24, 19, 21, 5, 7] for embedded systems. The majority of the work focuses on the compression of machinecode. A traditional approach partitions a program into segments. Before a portion of code in a segment is executed, the segment is decompressed into a buffer and then executed on the CPU or the VM as depicted in Fig. 6(a). The major disadvantage of this approach is that a memory buffer for execution is required that needs to be large enough to execute translation units of an application.

A new code compression technique [20] allows to execute bytecode in its compressed form 6(b). The fetch mechanism of the VM is extended to decompress bytecode on-the-fly when it is executed. This technique overcomes the disadvantage of a traditional approach by making the memory buffer for execution redundant. By carefully designing the decoder in the fetch mechanism of the VM Latendresse [20] et al. showed that the execution overhead for in-place decompression of Huffman-Code is at an acceptable speed. However, in [20] only the op-codes are compressed to Huffman-Code and the instructions are not split into different encoding streams for opcodes, literals, and addresses. This splitting is crucial for achieving good code compressions (see Experiments).

The integration of the decoder in the fetch mechanism of the VM [20] requires pre-computed decoder tables. The compressed bytecode is seen as a bitstream. A bitstream pointer points to the current fetch-position in the stream. The decoder reads the next k-bits from the stream and moves the bistream pointer k-bits to the right. The k-bits from the bitstream are used to decode the next code word from the stream. To speed up the decoding, a decoder table is used. The length of Huffman codes can become very long (up to 21 bits in our applications) and generating tables for these sizes is not viable in a WSN setting. To overcome this issue, we need to split the decoder table into sub-tables, which slows down the decoding. The sizes of the sub-tables depend on a space-speed trade-off.

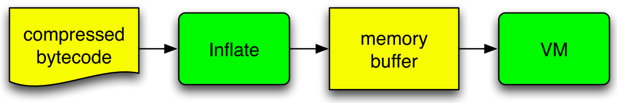
The decoder mechanism consists of several code fragments that are executed via the decoder table. The decoder starts executing a new code word in `decode0`. It reads the first k bits of the stream and uses the resulting number as an index for the first decoder sub-table. The sub-table contains the location of either a fragment that executes the op-code or more bits need to be processed from the stream to decode the word, i.e. another `decode<x>` fragment is executed. If a code-word was identified, more bits might have been read and the bitstream pointer is adjusted by `shift_left(<k>)`. We use following templates for decoding,

```
decode<x>: idx = get_nbits(<k>);
           goto *decoder_table<nr>[idx]; ...
code<y>:   shift\_left(<k>);
           <execute instruction y>; goto decode0; ...
```

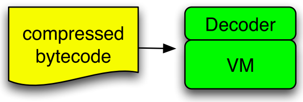
where `decode<x>` is for decoding parts of a code word and `code<y>` executes the op-code `<y>`.

To generate the decoder-tables we have a simple algorithm that determines the waste of a sub-table. A waste occurs if we have identical entries in the decoder table, i.e. with less bits the code words could be identified. Starting from the root node of the binary Huffman tree, we generate the sub-tables for the decoder. We increase the size of the tables until the waste threshold is saturated. We substantially simplified the decoder scheme of [20] that uses a complicated branch-and-bound scheme and canonical Huffman-codes.

As far as the author are aware of, we have the first instruction format for bytecode, that compresses integer



(a) Buffer Scheme



(b) Just-In-Time Decompression

Figure 6: Code Compression Model

literals, splits the instruction word into different streams depending on the op-code, and uses in-place compression. In particular, we had to make design decisions for encoding (1) literals that are operands (integers, floats, etc.), (2) jump addresses of conditional and unconditional jumps, (3) addresses of global variables, and (4) and function addresses of call statements.

In order to achieve a high compression rate we use alternating codes. I.e., the operands are encoded in a different code than their op-codes. Depending on the op-code, the decoder of the VM decides, which code is to be used to decode the operands of the instructions. For integer literals we use another Huffman code. Floating point literals are encoded as verbatim 32bit and 64bit codes respectively. Encoding jump addresses of conditional and unconditional jumps in bytecode turns out to be a hard problem: We decided to use relative addresses of fixed bitlength because they have the advantage of being position independent[3]. The reason why we did not choose a Huffman code is that Huffman codes do not have a monotonicity property, i.e. if code x is smaller than code y, than the length of the encoded code x is not necessarily smaller than the length of the encoded code y. Finding the distances between a jump target and a jump instructions is a non-trivial problem because in between there might be other jump instructions whose length depend on the current jump instruction. This results in a simultaneous discrete equation system that is very hard to solve. Therefore, we decided to encode the relative addresses of jumps as verbatim signed 16bit distance. Addresses of global variables and function addresses are encoded as 32bit words that are patched by the loader.

5. MACHINE- /BYTE-CODE SELECTION

In the following we discuss a discrete optimization problem that partitions the set of functions into two disjoint sets, i.e., the set of functions that are compiled to bytecode and the set of functions that are compiled to machine code. The optimization seeks for a partitioning, which has minimal energy consumption and which does not exceed the maximal available memory. We call this optimization problem the Machine-/Byte-code Selection(MBS) problem that arises with the capability of a virtual machine to seamlessly execute interpreted functions and functions in native machine code.

[3]For wireless sensor nodes that do not have memory management units, position independent code is an advantage if the memory needs to be re-organized (e.g., for code updates at runtime).

Compiled functions have a larger memory footprint than interpreted functions represented in compressed bytecode. However, compiled functions consume less energy than interpreted functions due to their interpretation overhead. Hence, there is a trade-off between energy and space, and we interested in a function partitioning that makes best use of the available memory to minimize the energy of the whole program. In this optimization problem, we consider as well transitions from the virtual machine to machine code and vice versa. These transitions between bytecode and machinecode are not for free in terms of energy because energy is used up for executing trampolines, rebuilding call stacks, and saving the state of the virtual machine. The transition overhead can be substantially, and if it is not modelled correctly, it could lead to sub-optimal answers.

Given a static call graph $G(F, C)$ where F is the set of functions and $C \subseteq F \times F$ is the set of call-sites of the input program. A call-site $(u, v) \in C$ represents a function call where u is the caller and v is the callee. The MBS problem may be stated as following mathematical program,

$$\begin{aligned} \min f = & \sum_{u \in F} \left[E_{vm}(u) x_u + E_{mc}(u)(1 - x_u)\right] + \qquad (1) \\ & \sum_{(u,v) \in F \times F} \left[E_{vm \to mc}(u, v) + E_{mc \to vm}(v, u)\right] x_u (1 - x_v) \\ \text{s.t.} & \sum_{u \in F} S_{vm}(u) x_u + S_{mc}(u)(1 - x_u) \leq S_{total} \qquad (2) \\ & x_u \in \{0, 1\}, \quad \text{for all } u \in F \qquad (3) \end{aligned}$$

where x_u is a binary variable expressing the decision whether a function $u \in F$ is represented as bytecode (i.e., x_u is one) or as machine code (i.e., x_u is zero). The objective function of the mathematical program in Eq. 1 is the amount of energy needed for a given partitioning and has energy parameters that model the energy consumption of functions and the transition between bytecode and machinecode. The parameter $E_{vm}(u)$ is the energy consumption of function u when executed in the virtual machine, the parameter $E_{mc}(u)$ is the energy consumption of function u when executed in machine code, $E_{vm \to mc}(u, v)$ is the energy consumption of switching from bytecode to machine at call-site (u, v), and $E_{mc \to vm}(u, v)$ is the energy consumption of switching from machine code to bytecode at call-site (u, v). The space constraint in Eq. 1 has space parameters for functions S_{vm} when executed in bytecode and S_{mc} when executed in machine code. The parameter S_{total} is the maximal available memory for the program and is determined by the available flash memory of the sensor node.

If a user manually overrides the representation of a function, the energy parameters of the function are adjusted accordingly, i.e., if the user specifies machine code the energy parameter for bytecode is set to a sufficient large number and if the user specifies byte code the parameter for machine code is set to a sufficient large number. Functions that are synthesized are excluded from the set F.

THEOREM 1. *The MBS problem is NP-hard.*

PROOF. See Appendix B. □

The energy function has a quadratic component for modeling the transitions between bytecode and machine-code. We linearize the quadratic terms of the objective function to solve the mathematical program with an integer linear programming solver. The term $x_u(1-x_v)$ of Eq. 1 is expanded in the objective function, and quadratic term $x_u x_v$ is replaced by variable $y_{uv} \in \{0,1\}$. We add linear constraints to the mathematical program to enforce the equivalence $y_{uv} = x_u x_v$ and obtain the following linear integer program for MBS:

$$\begin{aligned}
\min f = & \sum_{u \in F} [E_{vm}(u)x_u + E_{mc}(u)(1-x_u)] + \\
& \sum_{(u,v)\in F\times F} [E_{vm\to mc}(u,v) + E_{mc\to vm}(v,u)](x_u - y_{uv}) \\
\text{s.t.} & \sum_{u\in F} S_{vm}(u)x_u + S_{mc}(u)(1-x_u) \le S_{total} \\
& x_u \in \{0,1\}, \quad \text{for all } u \in F \\
& y_{uv} \le x_u, \quad \text{for all } (u,v) \in C \\
& y_{uv} \le x_v \\
& x_u + x_v - 1 \le y_{uv} \\
& y_{uv} \in \{0,1\}
\end{aligned}$$

The partitioning problem can also be seen as a multi-objective optimization problem of two dimensions, i.e., energy and space. In this formulation we are interested in the Pareto frontier. A point on the Pareto frontier is a bytecode- machine-code selection of functions that neither can improve code size without loosing energy and vice versa. The endpoints of the frontier are the solutions that all functions are either compiled to bytecode or to machine-code. An optimal solution of the MBS problem for a given memory limit S_{total} is Pareto efficient (i.e. is a point on the Pareto frontier). Note that the problem is highly discrete and therefore, the Pareto frontier is not continuous and might have gaps.

The pareto frontier is computed iteratively by initially setting the memory limit to a sufficiently large number resulting in a partitioning with a specific memory and energy consumption that gives an endpoint in the pareto frontier. In general, this partitioning will represent all functions in machine code. We compute the next point in the pareto frontier by using the space consumption of the previous point and decremented by one to determine the memory limit. This instance of MBS computes the next point in the pareto frontier. We compute this new instance and continue with the next point whose memory limit is determined by the previous one until no further points on the pareto frontier can be computed.

6. EXPERIMENTAL RESULTS

We conducted experiments to evaluate code compression and performance of our interpreter on the Intel iMote2 and for Atmega128-based motes. To investigate the robustness of our VM architecture we measured VM performance and compression rates for the SPEC CPU2000 benchmark suite (Spec2k) on the IA32 architecture. We compared performance and code sizes of TinyVM to other VMs proposed for sensor networks. We applied MBS and interpreter instruction set extensions to an elliptic curve cryptography and an image recognition benchmark to automatically generate all possible ASVMs along the energy/space Pareto frontier.

6.1 Interpreter Performance and Code Size

To conserve energy, motes are required to spend the major part of their lifetime in deep sleep. Prolonging the period of time a mote is awake (i.e., increasing the mote's duty cycle) consumes additional energy and reduces battery lifetime. For a WSN VM it is therefore of paramount importance to keep the interpretive overhead to a minimum.

Because of in-network processing, we tested TinyVM with large problem sizes. This would give us a good indication how our VM would scale to programs run on the more powerful upcoming mote generations such as the Intel iMote2. To begin with, we run the Spec2k testsuite on our VM on IA32. With Spec2k we would compile an *entire* application both to bytecode and to machine code and compare the execution times.

Fig. 7 shows the achieved VM slowdowns of interpreted bytecode over machine code plus the bytecode compression rates. Our slowdowns were between a factor of 11.4 (181.mcf) and a factor of 75.8 (183.equake). On average, TinyVM stayed below a factor of 38. These runs where conducted with the large Spec2k reference data sets. The upper half of Fig. 7 shows the relative sizes of compressed bytecode compared to machine

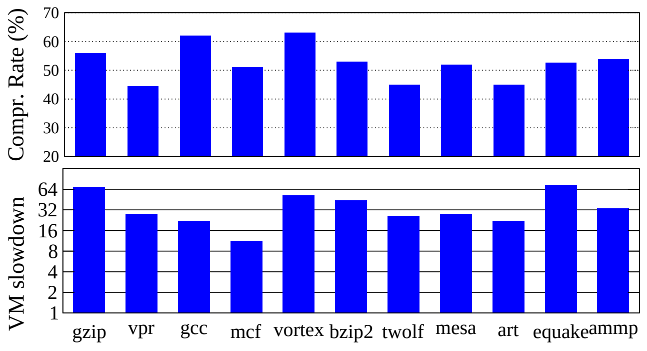


Figure 7: Spec2k Performance Figures

code.[4] TinyVM on IA32 is less than 30kB in size; machine code sizes of benchmarks were up to 602kB. Huffman-compressed bytecode yields codesize reductions of more than 50%. In-place execution of compressed bytecode therefore allows to store substantially larger programs on the mote or make more memory available to buffer data for in-network processing. Fig. 7 shows that TinyVM can execute even large programs like GCC in bytecode. This contrasts existing approaches where bytecode is limited to small sizes (128 bytes with Mate, and 1 user-defined function with SwissQM).

IA32 was indicative for architectures with a small register set: when enabling top-of-stack-caching (a VM optimization that caches the topmost stack element in a register), increased register pressure impacted performance in a negative way. When we repeated those experiments on the Intel iMote2, top-of-stack-caching increased performance. TinyVM's demand of two registers (one for the top-of-stack-pointer and one for the evaluation stack pointer) are much easier to get on RISC cores with a rich general-purpose register file than on irregular architectures.

On the Intel iMote2 under Linux, we ran the WSN elliptic curve cryptography (Ecc) implementation from [27] and the Susan benchmark from the MiBench embedded benchmark suite [13]. Like ECC, Susan is highly relevant for WSNs, because it performs various image recognition algorithms (in-network processing). On the Intel iMote2, we achieved a slowdown of 21.66 for ECC and 29.0 for Susan. We achieved space savings of 72.7% and 67.7% over machine code.

To evaluate TinyVM's performance on Atmega-based motes, we used 9 benchmarks from the WSN benchmark collection introduced in [31]. Results are depicted in Fig. 8. Therein, benchmark "sec.stt" performs a composition of benchmarks "stats" and "secure" as suggested in [31]. All other benchmarks are application building blocks from [31]. "Avrg." denotes the average slowdown and compression rate. Measurements were conducted on the cycle-accurate Avrora framework [36], with custom monitors to extract bytecode execution times, power consumption and memory usage. Like

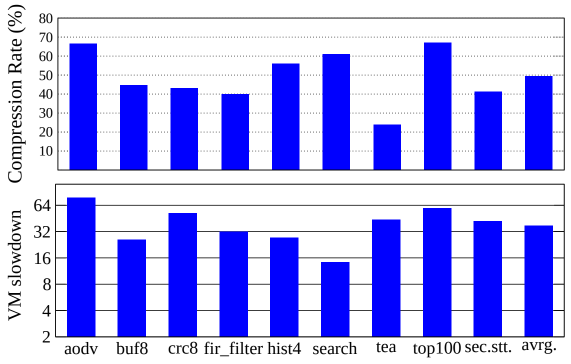


Figure 8: Sensebench Performance and Bytecode Compression Rates on the Mica2.

with previous benchmarks, we compiled each Sensebenchmark to Huffman-compressed bytecode and to machine code to compare the resulting execution times. As shown in Fig. 8, the slowdowns for interpreted compressed bytecode over machine code are between a factor of 14.4 and a factor of 72. Compressed bytecode was on average only 49.3% the size of machine code.

The TinyVM instruction set is synthesized to a given application such that only instructions occurring in the actual application are included with a TinyVM interpreter. For the Sensebenchmarks less than 20% of the instructions were needed on avarage. The resulting sizes of the TinyVM interpreters were between 4.3kB and 7kB, with an average of 5.4kB.

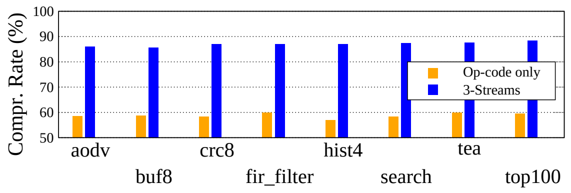


Figure 9: Compression Rates of Op-code only and 3-Streams vs. Plain Bytecode

Fig. 9 compares the compression rates that we achieved for the Sensebench benchmarks using opcode-only compression (of [20]) and compression of opcodes, literals and addresses (3-stream compression). For the surveyed benchmarks, 3-stream compression offers almost 20% better compression than opcodes only. 3-stream compression is thus advantageous for memory-constrained WSN architectures.

We compared TinyVM with other VMs for sensor net-

[4]In [4], off-line de-compression was used for C programs. Compressed code was inflated in memory prior to execution. This approach is not possible with WSNs due to the size of inflated code and the severe memory constraints of motes. This motivated in-place execution of compressed bytecode.

works, to investigate the performance that can be obtained in various parts of the sensor network VM design space (after all, TinyVM being a VM for nesC and C provides a relatively low abstraction level, which should lead to the highest performance in turn).

We instrumented the SwissQM [29] implementation on the Telos mote to determine execution speed and code size of SwissQM's user-definable bytecode functions (SwissQM is quoted as having the most space efficient program representation for sensor networks [6]). SwissQM provides comparatively simple bytecode functions without function-calls and arrays; we compiled a simple loop-based code to bytecode and compared its execution time to the execution time of the equivalent machine code. Bytecode execution on SwissQM was by a factor of 891 slower than machine code. A similar result is reported for an outlier detection application with DVM [1], where bytecode execution is by a factor of 107 slower than machine code. Although SwissQM achieves high code-density, Huffman-compression achieves between 40% to 50% higher space-savings over machine code than SwissQM. A similar result was observed for Mate [22]: a 128-byte code capsule for a discrete fourier transform was 32% larger than Huffman-compressed bytecode, and over 50% slower.

6.2 MBS: Machine-/Bytecode Selection

We conducted experiments for the MBS optimisation from Sec. 5 for two TinyOs benchmark programs: (1) the TinyOS elliptic curve cryptography implementation from [27] on the Atmega128 architecture with Mica2, and (2) an object recognition implementation based on the speeded-up robust features algorithm (SURF) [2]. Our framework draws no distinction between operating system and user code. Note the only functions that are not compiled to bytecode are functions that have inline assembler and for which, there does not exist a bytecode equivalent representation.

Our version of the TinyOS Ecc code consists of a total of 36 generated C-functions[5] and the SURF benchmark consists of a total of 23 generated C-functions. To automatically derive machine-code/bytecode partitions (i.e., ASVMs), our framework profiles the code to determine the energy consumptions of all functions u when executed in bytecode ($E_{vm(u)}$) and in machine-code ($E_{mc}(u)$). Moreover, we derive the code sizes for bytecode ($S_{vm(u)}$) and machine-code ($S_{mc}(u)$). For each pair of functions u, v we determine the overhead for the calls $E_{vm\to mc}(u,v)$ and $E_{mc\to vm}(u,v)$.

From the profiled data we compute the Pareto frontier for the MBS problem using CPLEX. The Pareto frontier consists of 303 points for Ecc, and it takes 6.1 seconds on

[5] The callgraph for this benchmark is depicted in Appendix C.

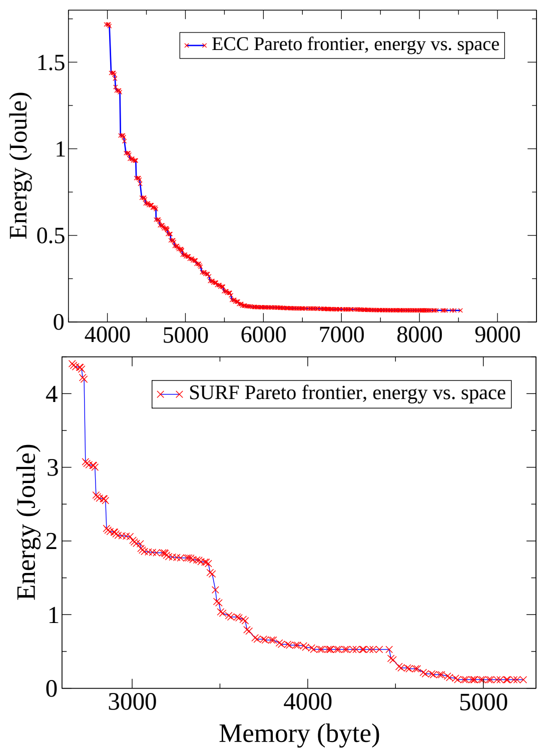


Figure 10: Pareto Frontiers.

a 2.3GhZ XEON E5345 Intel computer for computing the points with CPLEX. The pareto frontier for SURF on the same platform takes 1.5 seconds and consists of 140 points.

Fig. 10 shows the Pareto frontier for the whole Ecc and SURF code on Mica2. The x-axis gives the application image sizes, the y-axis depicts the energy consumption for a given machine-code/bytecode partition (called MBS partition in the following). Every point on the Pareto frontier represents an optimal MBS partition w.r.t. the given energy and space constraint.

Compiling the whole of Ecc and SURF to bytecode results in an Ecc application image of 3982 bytes and 2658 bytes, respectively, with a total bytecode execution energy bill of 1.717 J and 4.407 J. We get the other extreme point by compiling all of Ecc and SURF to machine code. This yields an Ecc application image of 8526 bytes and 5228 bytes, respectively, with a total bytecode execution energy bill of only 0.067 J and 0.117 J.

It follows from Fig. 10 that the Pareto frontier of the Ecc and SURF application has a steep gradient at the bytecode-side of the design space. This means that it is highly profitable to invest a few more bytes to compile certain small functions to machine-code. It is not unusual for applications to spend a major part of the execution time in a small fraction of the code, which means that this situation is likely to occur in practice.

It is worth to mention that SURF has a less smoother curve. This is due to the fact that functions in the SURF code have highly varying energy/memory consumptions.

Given the Pareto frontier, the WSN designer can select the most energy-efficient MBS partition for the available memory. Manual design space explorations would be infeasible: for Ecc with its 36 functions and for SURF with its 23 functions, the design space already contains 2^{36} and 2^{23} MBS partitions, respectively. Only 303 and 140, respectively, of those are Pareto-optimal. Automating the partitioning algorithm greatly facilitates the programmer's quest to come up with an energy-efficient solution for the application at hand.

Partition	Byte	Joule	Machine-code functions
1	3982	1.717699	n/a
2	3984	1.717696	task_function
3	4006	1.717692	task_function, f_do_all
4	4016	1.717548	task_function, p_iszero
5	4026	1.708419	task_function, ecc_memset
6	4048	1.437735	task_function, b_xor
7	4070	1.437731	task_function, b_xor, f_do_all
31	4532	0.675114	task_function, b_xor, ecc_memset, b_clear, b_copy, b_bitlength, f_add

Table 1: Machine-code functions of space-efficient MBS partitions.

For ASVM hardware abstraction, an ideal MBS partition will have functions at the bottom of the call graph assigned to machine-code. Thereby functionality close to the hardware is separated from higher-level bytecode, which facilitates hardware abstraction and later bytecode updates. Table 1 shows the top-most space-minimal MBS partitions for the Ecc application. Several small functions are put to machine-code, which might not be desirable for the above reasons (e.g., the task_funtion). With our automated MBS partitioning scheme, the programmer can attach the `vm` type specifier to such functions to override the proposed Pareto-optimal solution. With Ecc, pretty soon the functions at the bottom of the call-graph are assigned to machine-code (see, e.g., Partition 31 in Table 1). Thereby an advantageous partitioning in terms of hardware abstraction is achieved.

6.3 Instruction Set Extensions

To evaluate instruction set extensions, we employed a tight loop with only one function call:

```
for(i=0;i<MAX;i++) {
   call(arg0, arg1, ... , arg8);
}
```

During the first program run, `call` would be a binary function using the traditional native call interface; during the second run, `call` would be a VM instruction. We calibrated the above loop to execute for 10 seconds using the native call. On the iMote 2, the instruction set extension yielded a speedup of 12.7% over the native call. On the Mica2, the instruction set extension yielded a speedup by a factor of 2.16. This speedup is mainly due to the complicated calling convention of the Atmega128 CPU, which involves up to 9 8-bit register pairs and the program stack to accommodate procedure call arguments and return value. For a call from bytecode to machine code the arguments must be fetched from the VM evaluation stack and passed to the callee according to the callee's function signature (types of parameters and return type).

We evaluated instruction set extensions with the Ecc benchmark from the previous section. Once a partition on the Pareto frontier is selected, the WSN programmer can decide to make functions instruction set extensions of the VM. For example, by attaching the "`__is__`" type specifier to function `b_copy` with Partition 31 in Table 1, we make `b_copy` a VM instruction. Because `b_copy` is frequently called (9264 times in total, see the callgraph in Appendix C, the call-overhead of this MBS partition can be substantially reduced and performance increased by 2.4%[6] Likewise, we got a speedup of 3.7% for `bxor` and 4.59% for `b_bitlength`. Making a frequently executed function a VM instruction is highly profitable, but the function must be known to be a fixed and immutable part of an application (dynamic code update techniques can only update whole functions, but not part of a function). Instruction set extensions are similar to machine-code functions of application-specific virtual machines [23], but our mechanism saves the call overhead to machine-code functions and incorporates energy efficiency of functions to provide profile-guided, semi-automated creation of energy-efficient application-specific virtual machines.

7. CONCLUSION

We have introduced a virtual-machine based programming environment named TinyVM that uses compressed bytecode as its main code representation. Compressed bytecode is executed in-place, thereby avoiding CPU- and memory-intensive decompression on the mote. Bytecode interpretation can be further improved by the execution of machinecode. TinyVM seamlessly integrates the execution of compressed bytecode and machinecode, which maximizes performance while keeping the application's memory footprint small. Interpreter instruction set extensions save the procedure call overhead from bytecode to machine-code. Our Machine-code/Bytecode Selection algorithm (MBS) employs discrete optimization to achieve an optimal/near optimal partitioning of a WSN application into machine and bytecode. The

[6]We determine these call-overheads when we profile the application during MBS. The overhead for function v is $\Sigma_{u \in \text{callers}(v)} E_{vm \to mc}(u, v)$, where callers$(v)$ is the set of all functions that call v.

application developer can override this partitioning to explicitly trade space for speed/energy or vice versa[7].

We have implemented our programming environment and conducted experiments with WSN benchmarks on several platforms, including IA32, Intel iMote2 and Mica2. TinyVM is significantly more efficient than other approaches while maintaining a smaller memory footprint at the same time.

We are currently preparing a GPL-release of the TinyVM framework to make it available to the community and to get valuable feedback.

8. REFERENCES

[1] R. Balani, C.-C. Han, R. K. Rengaswamy, I. Tsigkogiannis, and M. Srivastava. Multi-level software reconfiguration for sensor networks. In *EMSOFT'06*, pages 112–121. ACM, 2006.

[2] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-up robust features (surf). *Comput. Vis. Image Underst.*, 110(3):346–359, 2008.

[3] J. Beutel. Fast-Prototyping Using the BTnode Platform. In *Proc. DATE'06*. ACM Press, 2006.

[4] B. Burgstaller, B. Scholz, and M. A. Ertl. An Embedded Systems Programming Environment for C. In *Proc. Euro-Par'06*, pages 1204–1216. Springer LNCS, 2006.

[5] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *PLDI'99*, pages 139–149, 1999.

[6] N. Costa, A. Pereira, and C. Serodio. Virtual Machines Applied to WSN's: The state-of-the-art and classification. In *ICSNC'07*. IEEE Computer Society, 2007.

[7] S. Debray and W. Evans. Profile-guided code compression. In *PLDI'02*, pages 95–105, New York, NY, USA, 2002. ACM Press.

[8] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks. In *SenSys'06*, pages 15–28. ACM Press, 2006.

[9] J. Ernst, W. S. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. In *PLDI'97*, pages 358–365, 1997.

[10] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[11] C. W. Fraser, E. W. Myers, and A. L. Wendt. Analyzing and compressing assembly code. *SIGPLAN Not.*, 19(6):117–121, 1984.

[12] D. Gay, P. Levis, R. v. Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proc. PLDI '03*, pages 1–11. ACM Press, 2003.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IISWC'01*, December 2001.

[14] D. R. Hanson and C. W. Fraser. *A Retargetable C Compiler: Design and Implementation.* Addison Wesley, 1995.

[15] M. Hempstead, N. Tripathi, P. Mauro, G. Wei, and D. Brooks. An Ultra Low Power System Architecture for Sensor Network Applications. In *Proc. ISCA'05*, pages 208–219. IEEE Press, 2005.

[16] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *Proc. ASPLOS-IX*, pages 93–104. ACM Press, 2000.

[17] K. Hong et al. Poster Abstract: TinyVM, an Efficient Virtual Machine Infrastructure for Sensor Networks. In *SenSys'09*. ACM Press, 2009.

[18] J. Koshy and R. Pandey. VMSTAR: Synthesizing Scalable Runtime Environments for Sensor Networks. In *SenSys'05*, pages 243–254. ACM Press, 2005.

[19] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *ICCS'94*, pages 270–277. IEEE Press, 1994.

[20] M. Latendresse and M. Feeley. Generation of Fast Interpreters for Huffman Compressed Bytecode. *Sci. Comput. Program.*, 57(3):295–317, 2005.

[21] H. Lekatsas and W. Wolf. SAMC: A code compression algorithm for embedded processors. *IEEE Transactions on CAD*, 18(12):1689–1701, 1999.

[22] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *ASPLOS-X*. ACM Press, Oct 2002.

[23] P. Levis, D. Gay, and D. Culler. Active Sensor Networks. In *USENIX/ACM NSDI'05*, May 2005.

[24] S. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In *Proc. Conf. on Advanced Research in VLSI*, 1995.

[25] J.-N. Lin and J.-L. Huang. A virtual machine-based programming environment for rapid sensor application development. In *COMPSAC'07*, pages 87–95, Washington, DC, USA, 2007. IEEE Computer Society.

[26] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *SIGMOD*, June 2003.

[27] D. J. Malan. Crypto for Tiny Objects. Technical Report TR-04-04, Harvard University, 2004.

[28] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. In K. Römer, H. Karl, and F. Mattern, editors, *EWSN*, volume 3868 of *LNCS*, pages 212–227. Springer, 2006.

[29] R. Müller, G. Alonso, and D. Kossmann. SwissQM: Next Generation Data Processing in Sensor Networks. In *CIDR'07*, pages 1–9. `http://www.cidrdb.org`, 2007.

[30] L. Nachman, R. Kling, R. Adler, J. Huang, and V. Hummel. The Intel Mote Platform: a Bluetooth-Based Sensor Network for Industrial Monitoring. In *Proc. IPSN'05*, page 61. IEEE Press, 2005.

[31] L. Nazhandali, M. Minuth, and T. Austin. Sensebench: toward an accurate evaluation of sensor network processors. In *IISWC'05*, pages 197–203, 6-8 Oct. 2005.

[32] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling Ultra-Low Power Wireless Research. In *Proc. IPSN'05*, page 48. IEEE Press, 2005.

[33] K. Römer, C. Frank, P. J. Marrón, and C. Becker. Generic Role Assignment for Wireless Sensor Networks. In *SIGOPS'04*, pages 7–12, Leuven, Belgium, Sept. 2004.

[34] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java&#8482; on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE'06*, pages 78–88, New York, NY, USA, 2006. ACM.

[35] Standard Performance Evaluation Corporation. Spec CPU 2000, 2000. http://www.spec.org/.

[36] B. Titzer, D. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN'05*, pages 477–482, 2005.

[7]There is also a tradeoff between speed and hardware abstraction involved; a bytecode-only setting is of course provided.

APPENDIX

A. AMPL MODEL

```
# define nodes and edges of static callgraph
# nodes are functions and edges are calls
set F;
set C within (F cross F);
# memory limit for program in bytes
param MemoryLimit, >=0;
# space and energy requirement of a function
# in bytecode representation
param SpaceVM{F}, >=0;
param EnergyVM{F}, >=0;
# space and energy requirement of a function
# in machine code representation
param SpaceMC{F}, >=0;
param EnergyMC{F}, >=0;
# energy requirements for a transitions between
# bytecode and machine code.
param EnergyVMtoMC{C}, >=0;
param EnergyMCtoVM{C}, >=0;
# binary decision variable for each function that
# decides whether the function is represented in
# bytecode or in machine code.
var x{F}, binary;
# Variable y[u,v] is one if both adjacent
# functions of edge (u,v) in the call graph
# are in bytecode; otherwise zero.
var y{C}, binary;
# minimize the energy of the whole program
# considering the energy of each function
# and the transitions between bytecode and
# machine-code.
minimize energy:
 (sum{u in F} (EnergyVM[u] * x[u] +
              EnergyMC[u] * (1- x[u]))) +
 (sum{(u,v) in C} EnergyVMtoMC[u,v]*(x[u] - y[u,v])) +
 (sum{(u,v) in C} EnergyMCtoVM[u,v]*(x[v] - y[u,v]));
# memmory constraint
subject to memory:
 sum{u in F} (SpaceVM[u] * x[u] +
  SpaceMC[u] * (1 - x[u])) <= MemoryLimit;
# linearization of term y[u,v] = x[u]*x[v]
# for all edges (u,v) in callgraph
subject to linear1{(u,v) in C}:
    y[u,v] <= x[u];
subject to linear2{(u,v) in C}:
    y[u,v] <= x[v];
subject to linear3{(u,v) in C}:
    x[u] + x[v] - 1 <= y[u,v];
```

B. PROOFS

Proof. Proof of Theorem 1. Given an instance of a $0-1$ knapsack problem with set of items A, a capacity of the knapsack K, and *profit* and *size* parameters for each item in set A. The reduction from 0-1 knapsack to MBS works as follows: Each element in A becomes a function u in F. There are no call-sites in the callgraph, i.e., $C = \emptyset$, and we set the parameters of the MBS problem as follows:

$$\begin{aligned} E_{vm}(u) &= 0, && \text{for all } u \in A \\ E_{mc}(u) &= \mathit{profit}(u), && \text{for all } u \in A \\ S_{vm}(u) &= \mathit{size}(u), && \text{for all } u \in A \\ S_{vm}(u) &= 0, && \text{for all } u \in A \\ S_{total} &= K \end{aligned}$$

We obtain a reduced mathematical program of MBS

$$\min f = \sum_{u \in A} \mathit{profit}(u) - \sum_{u \in A} \mathit{profit}(u) x_u$$

$$\text{s.t.} \sum_{u \in A} \mathit{size}(u) x_u \le K$$

that solves the Knapsack problem since the objective function can be rewritten to $\max f = \sum_{u \in A} \mathit{profit}(u) x_u$. $\square$

C. CALLGRAPHS

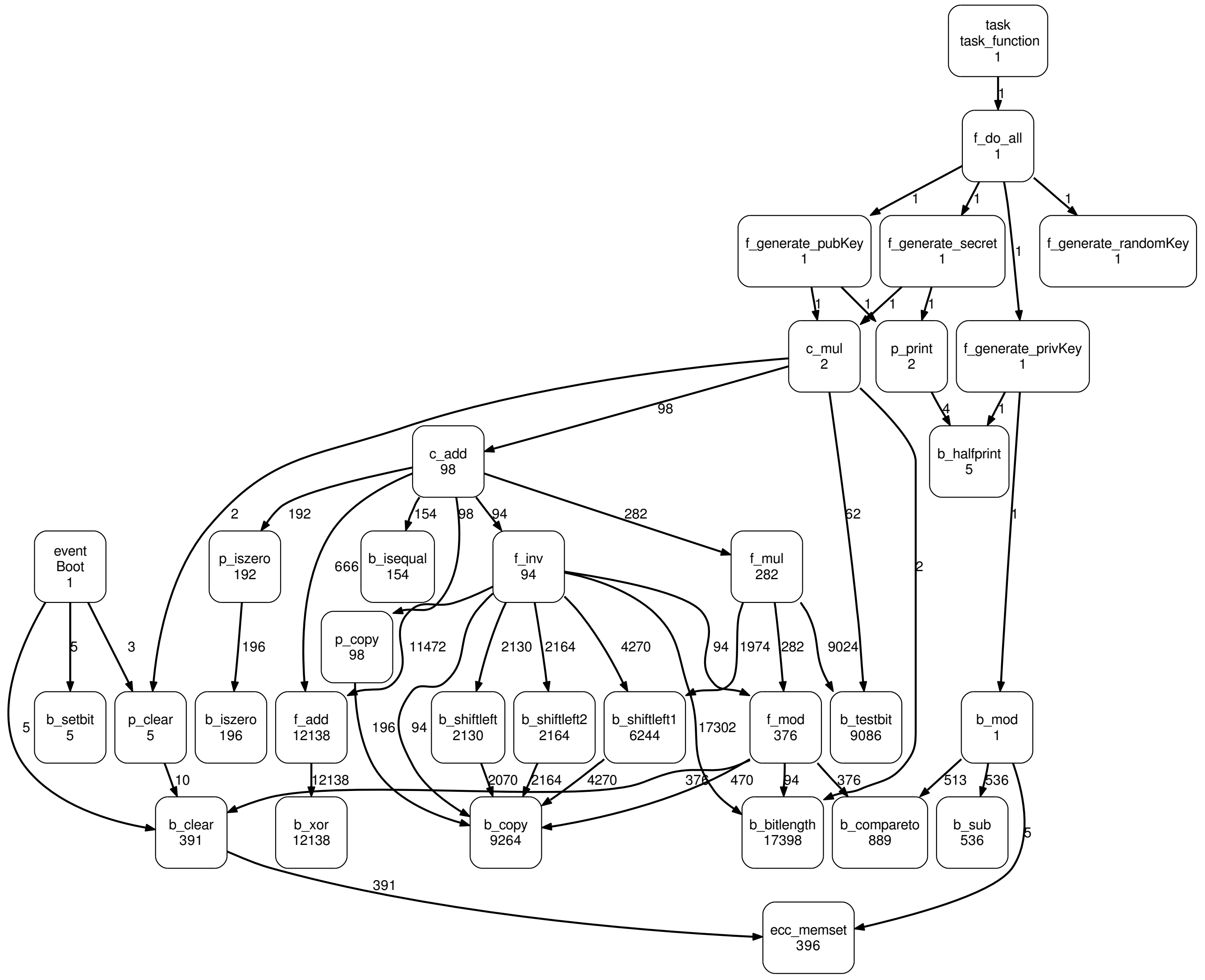


Figure 11: Callgraph for the Ecc benchmark: node and edge labels represent calling frequencies; three initialization functions have been omitted for space considerations. The Ecc callgraph was used with the MBS problem in Section 6.2. In Section 6.3, functions `b_copy`, `b_xor` and `b_bitlength` were compiled to VM instruction set extensions to save the call overhead from bytecode to machine-code functions.