Rethinking Java Call Stack Design for Tiny Embedded Devices

Faisal Aslam
Punjab University College of Information Technology (PUCIT)
Allama Iqbal Campus, Lahore, Pakistan
faisal.aslam@gmail.com

Ghufran Baig Mubashir Adnan Qureshi
Lahore University of Management Sciences (LUMS)
DHA, Lahore 54792, Pakistan
gufibaigi@gmail.com
mubashirmaq@gmail.com

Zartash Afzal Uzmi
Lahore University of Management Sciences (LUMS)
DHA, Lahore 54792, Pakistan
zartash@lums.edu.pk

Luminous Fennell Peter Thiemann
University of Freiburg
Freiburg 79110, Germany
fennell@informatik.uni-freiburg.de
thiemann@informatik.uni-freiburg.de

Christian Schindelhauer Elmar Haussmann
University of Freiburg
Freiburg 79110, Germany
schindel@informatik.uni-freiburg.de
haussmann@informatik.uni-freiburg.de

Abstract

The ability of tiny embedded devices to run large feature-rich programs is typically constrained by the amount of memory installed on such devices. Furthermore, the useful operation of these devices in wireless sensor applications is limited by their battery life. This paper presents a call stack redesign targeted at an efficient use of RAM storage and CPU cycles by a Java program running on a wireless sensor mote. Without compromising the application programs, our call stack redesign saves 30% of RAM, on average, evaluated over a large number of benchmarks. On the same set of benchmarks, our design also avoids frequent RAM allocations and deallocations, resulting in average 80% fewer memory operations and 23% faster program execution. These may be critical improvements for tiny embedded devices that are equipped with small amount of RAM and limited battery life. However, our call stack redesign is equally effective for any complex multi-threaded object oriented program developed for desktop computers. We describe the redesign, measure its performance and report the resulting savings in RAM and execution time for a wide variety of programs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (Garbage Collection)

General Terms Algorithms, Design, Experimentation, Performance

Keywords Wireless Sensor Networks, Call Stack, Memory Management, JVM, Java Virtual Machine, TakaTuka

LCTES 2012 June 12–13, 2012, Beijing, China.

1. Introduction

Advancements in hardware and economies of scale have led to the availability of inexpensive wireless sensor motes. The motes usually offer only a small amount of memory, limited computation power, and short battery lifetime. Even with all these limitations on an individual mote, these motes create a powerful wireless network when deployed in large number and embedded deeply within large-scale physical systems. Such a wireless network enables a wide variety of applications progressively envisioned by sensor network researchers. Some of such applications are environment monitoring, military surveillance, forest fire monitoring, industrial control, intelligent agriculture, and robotic exploration.

A common way of programming an application into wireless sensor motes is by using a low level or a specially designed programming language such as Assembly, C, or NesC [10]. These languages usually produce code that is difficult to debug, extend, reuse and maintain. To avoid the pitfalls of programming in a low-level language, there is a growing interest to program the motes using Java, a widely used high level programming language with a large developer community. Towards this end, several Java Virtual Machines (JVMs) for motes have been rolled out in recent years [3, 6, 13]. The advantages of using Java include portability, type safety and run-time garbage collection. However, Java—originally designed for 32-bit processors—has inadequate memory management and instruction set support for the 16-bit or 8-bit processors that are mostly used in wireless sensor motes. This paper proposes a new call stack design for Java from the standpoint of tiny embedded devices (e.g motes) with smaller than 10 KB of RAM. Our call stack redesign applies to all of the well-known JVMs for tiny motes which are stack based and use interpreters rather than Just-in-Time (JIT) compilers [3, 6, 13]. These JVMs avoid the use of JIT compilation as it requires a relatively large amount of memory—a scarce resource in motes—for storing and dynamically producing the native code at run time [12]. Furthermore, many motes follow the Harvard architecture requiring the JIT compiler to generate the native code in flash memory which has a long write time, resulting in a slower program execution [24]. This paper also assumes a JVM with a stack-based architecture, which

results in a smaller Java binary [20], compared to register-based architectures, and is used by most popular JVMs for motes [3, 6, 13].

We present CVCS (*Compile-time Variable Chunk Scheme*) and VSS (*Variable Slot Scheme*): two techniques that lead to a new stack design which is efficient in RAM and CPU utilization, without requiring changes in the application programs. The following provides a description of a typical function call stack design while identifying the causes of resource wastage in that design.

1.1 General Call Stack Design

A call stack stores a chain of function frames which represents the state information of functions under execution. The size of a function frame is computed during program compilation [15]. A new frame is pushed on the call stack when the function is actually invoked during run time. The frame is destroyed (i.e., popped from the call stack) when the function completes its execution [15]. For embedded devices, this design leads to three sources of RAM and CPU cycle wastage as is outlined in the following subsections.

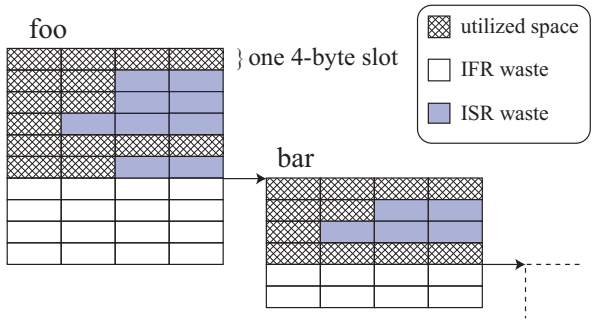


Figure 1. An example depicting two sources of RAM wastage.

1.1.1 Intra-frame RAM (IFR) waste

For each function, the frame size is computed during compilation of the program such that the frame can accommodate all the local variables as well as the maximum size of the operand stack required for the execution of that function [15, 20]. Thus, the space reserved for a function frame is the maximum that is ever needed during the execution of that function. While the space reserved for a frame is fixed, the actual *utilization* of the frame varies as the function execution progresses. It implies that the instantaneous stack utilization for a function `foo`, at a point when it invokes another function `bar`, may only be a fraction of the maximum allocation, as indicated in Fig. 1. On invocation of function `bar`, a corresponding frame is created and the execution of function `foo` is put on hold until after the function `bar` returns. During the execution of function `bar`, any unutilized frame space in `foo` remains untouched and thus wasted. This waste, which we call the Intra-Frame RAM (IFR) waste, can quickly accumulate when the program is implemented using several nested functions. We present CVCS in Section 2 that significantly reduces the IFR waste.

1.1.2 Frame allocation time (FAT) waste

The memory for a function frame may be allocated when that function is invoked and deallocated when the execution of that function is completed (see Java specifications [15] Section 3.6). Memory allocation is known to be a slow operation as it usually requires searching for a memory block large enough to accommodate the complete frame. This slow operation has to be carried out a number of times during the execution of a program for each function invocation, thus it wastes significant CPU cycles. We refer to this as Frame Allocation Time (FAT) waste.

To eliminate FAT waste and to make a Java program run faster, a JVM designed for desktop computer may allocate a large portion of RAM for the call stack in advance instead of allocating RAM for each individual frame. Thus, a new allocation will be required only when that large portion of RAM is almost used and is no longer able to accommodate a new frame. Although pre-allocating a large-sized memory block for the call stack is useful on a desktop computer, a JVM designed for motes should reserve memory more carefully. This is to avoid a possible overbooking of the limited amount of available RAM that is shared between the call stack and the heap. Furthermore, a JVM for motes should also minimize FAT waste to maximize battery lifetime. CVCS achieves both of the above mentioned objectives, that is it reduces FAT waste without allocating an arbitrarily large portion of RAM for the call stack.

Figure 2. IRIS Mote: An IRIS mote used in our experimental evaluation. The mote has 128 KB of flash, 8 KB of RAM, and it uses an 8-bit `AtMega128L` processor. It is powered by two AA batteries and is equipped with a 2.4 GHz radio for communication with other motes.

1.1.3 Intra-slot RAM (ISR) waste

Another source of wasted RAM in the Java call stack stems from the use of fixed allocation slots. A slot is an atomic unit of a frame, which can hold at most one data item—from operand stack or from local variables. A data item itself might occupy multiple slots. Standard Java uses 4-byte slots so that data may be efficiently accessed on a 32-bit processor[1]. When a data item of size smaller than 4 bytes is stored in a 4-byte slot, some portion of that slot remains unused and thus wasted as shown in Fig. 1. We refer to this as Intra-Slot RAM (ISR) waste.

Wireless motes are usually equipped with an 8-bit or a 16-bit processor. Thus, without compromising performance, a JVM may save RAM on a mote by using smaller than 4-byte slot size. One of our design goals is to allow the JVM to dynamically select the slot size based on the processor type making it suitable to run on processors of varying word sizes. However, achieving this goal is challenging given that all the Java bytecode instructions support 32-bit or 64-bit operations [15]. A possible solution is to modify the instruction set for processors with another word length but this may render the modification unsuitable for processors with yet another different word length. Our design includes a novel solution, called Variable Slot Scheme (VSS), which enables a JVM to choose slot sizes depending upon the target processor.

In this paper, we present the design and evaluation of CVCS and VSS using TakaTuka JVM [3, 4]. We selected TakaTuka because it is open-source, mature enough to support a wide variety of benchmark programs and is designed for tiny wireless sensor motes. The Java benchmarks are evaluated on widely used IRIS motes (see Fig. 2) where each mote is equipped with only 8 KB of RAM [2]. When evaluated over a large number of benchmark programs of various types, our implementation of CVCS and VSS results in average RAM savings of about 30% and about 80% reduction in the memory operations.

[1] A 32-bit processor needs extra CPU cycles if a memory address other than a multiple of 32 is accessed.

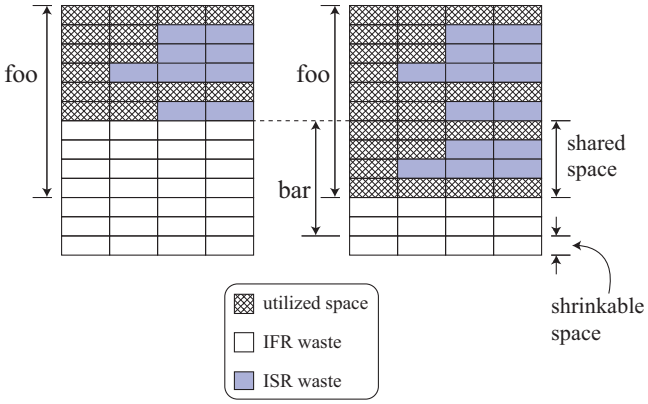


Figure 3. Reduction in the RAM waste using chunks that carry overlapping frames. The figure shows two instances in time of the same chunk during the execution of the program.

1.2 Contributions

In the context of our new stack design, we make the following contributions in this paper:

1. For the first time a compile-time redesign of the call stack with runtime shrinking is presented, for object oriented, multi-threaded programs. Contemporary work either uses a runtime redesign or is limited to procedural programming as explained in Section 5.
2. Use of a novel inter-procedural data flow analysis to calculate—during compile time—the size of a memory chunk needed to hold function frames in a nested call of given depth (see Figure 3).
3. A design that eliminates IFR waste by allowing overlapping memory blocks holding the frames of nested functions.
4. A design which allows a programmer to select a slot size that better matches the width of the hardware bus, thus bringing a reduction in ISR waste.
5. A thorough evaluation of our new stack design using around two dozen well-known benchmark programs in various categories.

1.3 Outline of the paper

The rest of the paper is organized as follows: We present Compile-time Variable Chunk Scheme (CVCS) in Section 2. Section 3 describes our Variable Slot Scheme (VSS). Benchmarks used to evaluate our schemes, evaluation setups and comprehensive results are given in Section 4. Related work is detailed in Section 5. Finally we draw our conclusions in Section 6.

2. Compile-time Variable Chunk Scheme

Using current Java Virtual Machine (JVM) specifications, existing implementations determine the size of a function frame at compile time using an intra-procedural data-flow analysis [15]. Subsequently, this pre-calculated size is used during the execution of the program—when a function is invoked—to actually allocate the memory for the function frame. This leads to FAT and IFR wastes. In contrast, the CVCS design creates chunks (instead of function frames) during the execution of the program; the size of a chunk is also pre-calculated during the compile time but the data-flow analysis to compute this size is inter-procedural and field-sensitive.

CVCS allows a single chunk to carry multiple frames leading to less frequent runtime memory allocations, which results in a reduction in FAT waste and, therefore, an increase in the lifetime of power-constrained devices. As our results indicate (Section 4), the steady state memory utilization using CVCS remains lower than the case when CVCS is not used, even though the size of a chunk is at least as large as a single function frame. This is because CVCS also exploits using the memory area that is *allocated to but not utilized by* a function frame during the time that function makes a nested call. In essence, this means a complete elimination of the IFR waste, except for the last frame of each chunk. A chunk is allocated—upon a function invocation—either when there exists no chunk in the memory or when the chunk at the top of the stack does not have enough memory space left to accommodate all the data items of the invoked function. A chunk is deallocated (i.e., popped from the call stack) if and only if it has no data item left.

```
1  public static void foobar(...) {
2      if (...) {
3        foobar(...);
4      }
5      ....
6  }
```

Figure 4. Offline data-flow analysis limitation in the presence of a recursive function.

The CVCS design circumvents the IFR waste (through frame overlaps) and the FAT waste (through pre-allocated chunks), and is best explained with an example shown in Fig. 3. The figure shows that the last four slots in the frame of `foo` are not utilized at the time when `foo` makes a nested call to function `bar`. These four slots are guaranteed to remain unused until `bar` returns. We use this fact to start storing the data items of function `bar` from the first unused slot in the chunk—a slot that would be used by function `foo` but only after function `bar` returns. Thus, `foo` and `bar` share those four slots that would otherwise remain allocated but unutilized. The figure shows only two functions sharing slots with each other but in complex programs, multiple functions could share the same set of slots.

This sharing in CVCS reduces the IFR waste and makes additional RAM available to a program, such that larger programs can be facilitated to run on memory-constrained devices. Figure 3 also illustrates the boundary case when two slots at the end of function `bar` remain unused (assuming that the frame of a function invoked by `bar` can not fit into the remaining slots of the chunk). Any space in the current chunk, after the frame of `bar` function, is also unutilized but is shrinkable, as we describe in Section 2.2.

2.1 Chunk size calculations

The compiler for TakaTuka has built-in Data-Flow Analysis (called OGC-DFA) that is inter-procedural context sensitive, field sensitive and flow sensitive [4]. During the compilation of a program, we use OGC-DFA to construct the method call graph and determine the size of a frame needed on each instruction. This information is fed into the following equation to calculate the size of chunk for function f, given a depth d of nested method calls originating for f:

$$C_{f,d} = \begin{cases} \max(F_f, \max\limits_{i}(F_{f,i} + \max\limits_{g}(C_{g(f,i),d-1}))) & \text{if } d > 0, \\ 0 & \text{if } d = 0. \end{cases} \quad (1)$$

In the above equation, F_f is the total size of a frame of the function f. $F_{f,i}$ is the maximum size of function f's frame at instruction i and $g(f,i)$ is the function g invoked by the instruction i of the function f.

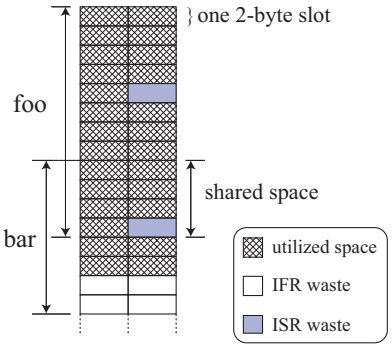


Figure 5. Reduction in the RAM waste (in addition to the one shown in Figure 3) using VSS.

2.2 Shrinking chunk size

The OGC-DFA takes into account all the functions that could be invoked and instructions that could be executed at runtime in the worst possible scenario. At runtime a subset of the methods analyzed by OGC-DFA are invoked and a subset of instructions are executed. The lack of exact information about a program, during its compilation affects the outcome of Equation 1, as the chunk size calculated using it could be larger than the RAM actually needed at runtime. To overcome this shortcoming, we shrink the size of current chunk before creating a new chunk. Recall that a new chunk is created only when the current chunk does not have sufficient space left to accommodate all the data items of a newly invoked function. Before creating such new chunk, we shrink any unused space available in the current chunk that is not part of the last frame of the chunk. Fig. 3 shows one slot at the end of the chunk that will be removed when a new chunk will be created because that slot is not part of the last frame of the chunk and the three empty slots at the end of the chunk are not sufficient to accommodate the next frame. The exact process of shrinking chunk size may differ for each JVM depending upon its implementation details. In the TakaTuka JVM, the shrinking requires updating a free-list, to make extra slots available for future use.

2.3 Selection of depth

The depth d in Eq. 1 determines how many levels of nested function calls are to be accommodated in a single chunk. To save RAM, the depth d should be selected carefully. A small value of d will result in a higher IFR waste and more frequent chunk allocations, whereas a large value of d will cause RAM overbooking increasing the likelihood of a heap overflow. Furthermore, a large value of d will also increase the probability of error in the chunk size calculation based on Eq. 1. For example, assume that `foobar` in Fig. 4 might be called at most n times during the program execution. Given that the value of n is unknown during the compilation of program, the $C_{foobar,d}$ for $d > n$ will return a chunk size greater than actually needed for the execution of the program. In conclusion, the depth d should be selected carefully — neither too large nor too small.

3. Variable Slot Scheme

A frame is composed of multiple fixed-size slots [6, 15] each of which can hold at most one data item. This design facilitates constant-time lookup of any data item by indexing into the frame. In Java, the maximum number of slots required by the frame of a function is calculated during compilation. These numbers are stored in the Java binary and are used for memory allocation during program execution. Standard Java has 4-byte slots as Java was originally designed for 32-bit processors. There are two implications of having 4-byte slots:

- Each data item in a frame is always stored at a memory offset that is a multiple of 32, leading to a fast data access on a 32-bit processor. A 32-bit processor needs extra CPU cycles if a memory address other than a multiple of 32 is accessed.
- A data item smaller than 32 bits[2] is also stored in a 32-bit slot, leading to waste of precious RAM space.

For a 32-bit processor, the slot size of 4 bytes presents a trade-off between execution speed and total RAM consumption. A wireless sensor mote usually has either an 8-bit or a 16-bit processor. For such processors, there is no slow-down as long as the slot size is greater or equal to the processor's word size. Therefore, the use of smaller than 4-byte slots on a wireless sensor mote can result in RAM savings without undermining the speed of bytecode execution.

This section presents *Variable Slot Scheme* (VSS) that offers an option to the programmer to select from 8, 16, and 32-bit slot sizes at compile-time. The resulting bytecode uses fixed size slots at run time maintaining the constant time access to the data stored in slots. VSS takes a standard Java binary (the `class` file generated for 32-bit machines) and transforms it into another Java binary that uses the slot size selected by the programmer. To accomplish this, VSS needs to extend the Java bytecode instruction set by introducing new instructions that support VSS. The following section discusses the design of VSS in detail.

3.1 VSS Instruction Design

The standard Java bytecode instructions support only 32-bit or 64-bit operations [15]. In Sun JVM, data of size smaller than 32 bits is sign extended to a 32-bit integer before being stored in a local variable or on the operand stack of a function [15]. Subsequently, any operation carried out on that data is the same as that for a 32-bit integer. Therefore, introducing VSS in the TakaTuka JVM requires extending the Java bytecode instruction set to support smaller than 32-bit data types.

The JVM specification defines only 204 out of 256 possible opcodes for bytecode instructions [15]. The VSS could use those $256-204 = 52$ available opcodes to create customized instructions for supporting smaller than 32-bit operations. However, the size and complexity of the JVM increases with the number of instructions it supports. Therefore, in order to fit the JVM on a mote's flash a conservative approach in the extension of Java bytecode instruction set is required. We have considered two possible extensions of the Java bytecode to support the VSS. These extensions are explained using a simple example.

Example: Fig. 6 shows Java source of a simple function `addTwo` that returns the sum of two input variables: a 16-bit `short` and an 8-bit `byte`. The Java binary of function `addTwo` generated by the standard Java compilation is shown in Fig. 7. The frame of the function `addTwo` requires five 32-bit slots, including three slots for local variables requiring $3 \times 32 = 96$ bits of RAM, and two slots for operand stack requiring $2 \times 32 = 64$ bits of RAM.

3.1.1 Specialized Operations and Data Access

This extension introduces a new instruction for a smaller than 32-bit data type corresponding to each existing instruction of 32-bit and 64-bit data types. For example, corresponding to the standard Java instruction `IADD`, a new instruction `SADD` is introduced for adding two 16-bit `shorts`. Fig. 8 shows the Java binary with extended instruction set for function `addTwo`. The number of slots required by the function, using VSS, depends upon the slot size

[2] In Java, data types `short`, `byte`, `boolean` and `char` use less than 4 bytes.

```
1   public static int addTwo(short s, byte b) {
2       int i = s + b;
3       return i;
4   }
```

Figure 6. Source code of function `addTwo` that adds a `short` and a `byte`.

```
1       ILOAD_1
2       ILOAD_2
3       IADD
4       ISTORE_3
5       ILOAD_3
6       IRETURN
```

Figure 7. Java binary of function `addTwo`.

```
1       LOAD_SHORT_CHAR  1
2       LOAD_BYTE_BOOLEAN  2
3       BYTE2SHORT
4       SADD
5       SHORT2INT
6       ISTORE_3
7       ILOAD_3
8       IRETURN
```

Figure 8. Java binary of function `addTwo` with "Specialized operations and data access" described in Section 3.1.1.

selected by the programmer during creation of Java binary. If 8-bit slot size is selected then the local variables will need seven slots (i.e., $7 \times 8 = 56$ bits of RAM) and the operand stack will require four 8-bit slots. In contrast, in case 16-bit slot size is selected then the local variables will need four slots (i.e., $4 \times 16 = 64$ bits of RAM) and the operand stack will require two 16-bit slots. Thus the RAM consumption is reduced significantly with smaller slot sizes.

The main drawback of the *"Specialized operations and data access"* extension is that it will increase the size and complexity of JVM significantly. This is because Java has dozens of standard instructions for 32-bit and 64-bit operations (e.g., `IADD`, `IMUL`, `IAND`) and this scheme will require creating 8 and 16-bit versions of those instructions.

The Darjeeling JVM [6] has a bytecode extension for supporting 16-bit data types that is similar to the extension presented here. Unlike the goal set for the VSS, the Darjeeling JVM does not have the option to choose slot size during the program compilation as slot size for Darjeeling programs is always equals to 16-bit. The 16-bit slot size could lead to the reduction in the performance of JVM on a 32-bit processor and ISR waste on an 8-bit processor.

3.1.2 Specialized Data Access Only

A better way to implement VSS is to employ the existing instructions provided by standard Java for all 32-bit operations but design a new set of customized 8-bit and 16-bit instructions for storing and retrieving information from operand stack and local variables. If a standard 32-bit operation requires a data item that is stored in less than 4 bytes then use casting instructions to convert data into 32 bits before applying the standard 32-bit operation. It implies that besides introducing new customized instructions for loading, storing and casting data, the rest of the Java bytecode instruction set will remain unchanged. We have carefully designed *six* new bytecode instructions that can be used to implement VSS for different types of processors. These new instructions are listed in Table 1. The increase in the JVM size to support VSS is small because it only needs to support six new instructions. Furthermore, the JVM will be suitable for different kinds of processors and save RAM without any additional change in it. Therefore, we have selected *"Specialized data access only"* to extend the TakaTuka JVM and for detailed evaluation. The Java binary of function `addTwo` with the TakaTuka VSS extension is shown in Fig. 9.

Both of the above bytecode extensions introduce new instructions for moving data between the operand stack and the local variables. For example, both extensions need a `LOAD_SHORT_CHAR` instruction to load a 16-bit value onto the operand stack. However, unlike the first extension, the TakaTuka VSS extension does not duplicate all the 32-bit `integer` operations (there is no `SADD` in Fig. 9). This significantly reduces the number of additional instructions required to support the VSS.

Instruction	Description
`LOAD_SHORT_CHAR index`	Loads data of type `short` or a `char` on the operand stack of a function.
`STORE_SHORT_CHAR index`	Stores data of type `short` or a `char` in a local variable of a function.
`LOAD_BYTE_BOOLEAN index`	Loads a data of type `byte` or a `boolean` on the operand stack of a function.
`STORE_BYTE_BOOLEAN index`	Stores data of type `byte` or a `boolean` in a local variable of a function.
`CAST_STACK_LOCATION index type`	Casts a given stack location to `integer`.
`METHOD_STACK_LOCATION index from-type to-type`	Casts a given stack location with given type to another type.

Table 1. Set of additional bytecode instructions in the TakaTuka VSS Extension.

3.2 Changing Bytecode for Supporting VSS

Given the source code of a program, the TakaTuka JVM creates Java binary in two steps: (1) it uses standard Java compilation for the creation of the `class` files from the program source code, (2) it transforms the `class` files into the TakaTuka binary, called Tuk [3]. The Tuk file is optimized for efficient RAM and flash usage. The Tuk file creation and optimizations are performed on a desktop computer before the application is transferred to a mote for execution. Therefore, the optimization process does not consume resources of a mote. We have extended offline compilation of the Tuk file with VSS to decrease RAM consumption. The VSS updates the bytecode during the creation of the Tuk file based on the selection of the slot size by the programmer. We now describe how VSS updating of the bytecode is carried out.

3.2.1 Reducing Input Parameters Size

The local variables of a function consist of its *input parameters* and the variables defined within the body (called *body-variables*). For example, the function addTwo in Fig. 6, has three local variables including two input parameters s, b and one body-variable i.

The type information of the input parameters is included within the Java binary. Using this information, the Java binary is updated, by replacing 32-bit load and store instructions with corresponding smaller than 32-bit instructions. That is the first instruction `ILOAD` of function `addTwo` in Fig. 7 is replaced by `LOAD_SHORT_CHAR` in Fig. 9 based on the input parameter type information.

3.2.2 Reducing Operand Stack Size

In order to determine the number of slots required by the operand stack, it is necessary to analyze the bytecode of the function. To this end, we developed the VSS Data-Flow Analyzer (VSS-DFA) that is an extension of the Bytecode Verification Data-Flow Analyzer (BV-DFA) [4, 15].

The Java bytecode is verified using BV-DFA and is then input to the VSS engine. The VSS engine modifies this already verified

```
1       LOAD_SHORT_CHAR  1
2       LOAD_BYTE_BOOLEAN  2
3       CAST_STACK_LOCATION  0  BYTE
4       CAST_STACK_LOCATION  1  SHORT
5       IADD
6       ISTORE_3
7       ILOAD_3
8       IRETURN
```

Figure 9. Java binary of function `addTwo` with TakaTuka VSS extension: "Specialized data access only".

bytecode by adding load and store instructions for smaller than 32-bit data types. Subsequently, the engine runs VSS-DFA to reanalyze the updated bytecode. If now any verification errors in the changed bytecode are discovered, they could only be due to the newly added smaller than 32-bit load and store instructions. These errors are corrected by inserting typecast instructions in the bytecode of the function. The VSS-DFA also calculates the reduced frame size for each function based on newly added smaller than 32-bit instructions. Several additional enhancements needed in BV-DFA to realize the VSS-DFA are part of the implementation details and are omitted here.

3.2.3 Reducing the Size of body-variables

Unlike the input parameters, the type information for body variables is not included in the Java binary. Therefore, their size must be computed during data-flow analzsis. The TakaTuka VSS engine uses a simple algorithm to estimate the types of body-variables. This algorithm works as follows:

1. During VSS the data flow analyzer computes the maximum size of the data that is ever stored in a body-variable.
2. The type of a body-variable is determined based on the maximum-sized data stored in it. For example, if a body-variable never stores data of size greater than 16 bits during analysis, then that variable should have either `char` or `short` type. The runtime semantics of both `char` and `short` are the same, so it is safe to choose any of these types for that variable.
3. Finally, the bytecode is changed on the basis of that type.

The above algorithm does not guarantee that the size of each body-variable will be minimized, however, the algorithm is still useful in saving precious RAM.

4. Results and Discussion

For a comparative evaluation of CVCS/VSS with a typical frame-based scheme, our selected benchmarks included programs that are either developed for sensor motes or are flexible enough to adjust their memory requirements based on program parameters. Table 2 shows our selected programs divided into five different categories. Programs in the first category are network-centric while those in the remaining four categories can be run on individual motes without requiring a networked setup. These programs vary from small single-threaded to large-sized with multiple threads [1, 19, 21, 22]. As our evaluation platform, we used IRIS motes (see Fig 2) which are equipped with only 8KB of RAM enforcing strict memory constraints [2, 14]. In case a program goes beyond the given RAM constraint, the TakaTuka JVM ceases its execution and throws an exception. For each program, we obtained results using different metrics described in the following subsection.

4.1 Performance Metrics

- **RAM Reserved**: The amount of RAM reserved by the call stack at a given point of execution of a program. This includes RAM that is currently in use at that point as well as the RAM reserved at that point for the possible future use. The main criteria of determining the effectiveness of a scheme should be its reduction of the RAM reserved, because the smaller the RAM reserved by the call stack the greater the chance that a mote will be able to accommodate a bigger and feature rich program. Furthermore, smaller RAM reserved for the call stack means that a large portion of RAM is left for the heap hence the heap can accommodate more and larger objects.
- **RAM Waste**: The RAM reserved by the call stack but not yet utilized. The RAM waste is a subset of RAM reserved. Unlike RAM reserved, smaller RAM waste alone is *not* sufficient to establish superiority and effectiveness of one scheme over another. However, a scheme should try to reduce waste to decrease the total RAM reserved and make more RAM available to a program.
- **Number of Allocations**: The total number of allocations for maintaining the call stack that occur during the execution of a program. This does not include the number of allocations on the heap as a scheme for the call stack has no effect on them. The reduction in the number of allocations and deallocations results in a faster program execution and increased battery lifetime of a mote.
- **Execution Speed**: We also measure the total time taken by each program to complete its execution and percentage reduction in that time when CVCS and VSS is used.

		Wireless Sensor Networks
1	CTP [11]	an address-free collection protocol
2	Dymo [7]	a on-demand routing protocol
		Network Algorithms
3	Dijkstra [1]	Link state routing protocol
4	Bellman-Ford	Distance vector routing protocol
5	Floyd-Warshall [1]	Shortest path in a weighted graph
6	Kosaraju [1]	Strongly connected component
7	Tarjan [1]	Least common ancestor of nodes
8	Edmonds [1]	Optimum branching algorithm
		Searching and Sorting
9	QuickSort	$0(NlogN)$ sorting algorithm
10	HeapSort	$0(NlogN)$ sorting algorithm
11	Red-Black Tree	Self balancing binary tree
		Maths and Engineering
12	MatrixMul [19]	$N \times N$ matrices multiplication
13	NQueen [19]	N queens placement problem
		Memory Benchmarks
14	GCBench [4, 21]	Designed to mimic memory usage behavior of real applications.
15	GCOld [4, 21]	GC benchmark developed by Oracle Inc
16	HeapTest [22]	Memory benchmark developed for SunSpot motes.
17	Benchmark [22]	Memory benchmark developed for SunSpot motes.

Table 2. Set of Benchmarks used for evaluation.

4.2 Single Machine Programs

All programs in our evaluation list, except for those in the first category in Table 2, can be executed on a single wireless mote without requiring a networked environment. For each of these programs, we measure the RAM reserved and RAM waste at the entry and exit of each function invocation.

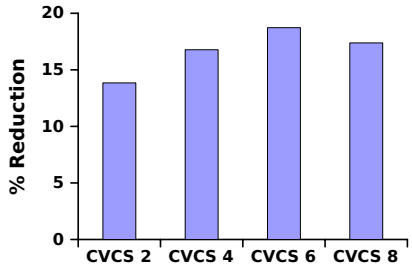


Figure 10. CVCS with Shrinking: Average percentage reduction in RAM reserved by the call stack, compared to a frame-based scheme. In the figure, CVCS 2 means CVCS with depth $d = 2$.

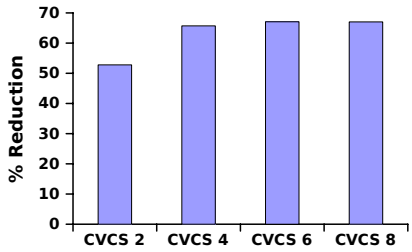


Figure 11. RAM Waste: Average percentage reduction in the RAM wasted by the call stack with the use of CVCS (compared to a frame-based scheme).

4.2.1 CVCS

Fig. 10 shows the average percentage reduction in the RAM reserved by the call stack of a program with the CVCS compared to a frame-based scheme. The results are shown with different depths d (see Eq. 1). The average RAM waste during the execution of each program is shown in Fig. 11 and average number of memory allocations is shown in Fig 12. We make the following observations and inferences from these graphs:

- CVCS always performs better (in terms of RAM reserved) compared to a frame-based scheme even for small d.
- The savings in RAM reserved increases with d but on IRIS mote $d = 6$ is sufficient to realize the maximum gains (for the set of benchmarks evaluated). The average percentage reduction in RAM reserved using CVCS with shrinking is 18.72% with $d = 6$. A further increase to $d = 8$ results in less memory reduction, because some requests for allocating big chunks were not fulfilled during the execution of the benchmarks. If insufficient memory is available to create a chunk for multiple methods, a smaller chunk to fit a single method is created, deteriorating the result for $d = 8$.
- The saving in RAM reserved comes with a reduction in the number of allocations and deallocations required to maintain the call stack. Fig. 12 shows that using CVCS, with $d = 8$ and shrinking, requires on the average about 80% fewer allocations to maintain the call stack as compared with the typical frame based scheme. Furthermore, the number of allocations in CVCS remains small no matter what d is selected.

4.2.2 Variable Slot Scheme

We now measure the performance of the Variable Slot Scheme (VSS) with slot size of 1 and 2 bytes. The evaluation of VSS is carried out using two kinds of observables:

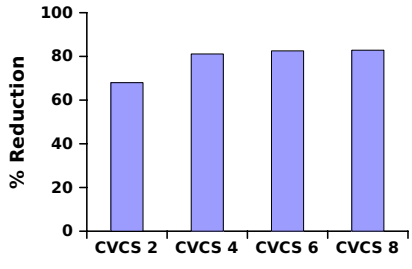


Figure 12. Number of Allocations: Average percentage reduction in the number of allocations achieved by using CVCS.

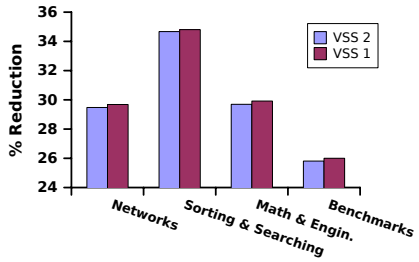


Figure 13. Average percentage reduction in the size of frames by the VSS. These results are based on reduction calculated in the frame size during the *compilation* of the programs.

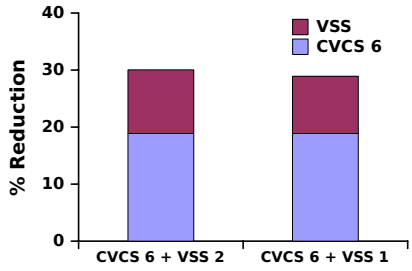


Figure 14. Aggregated average percentage reduction in the RAM reserved by the call stack with the VSS and CVCS, during the *execution* of the programs.

1. The average percentage reduction in the frames size at compilation.
2. The average percentage reduction in the RAM reserved for call-stack during program execution.

The Java binary includes information about the maximum frame size needed during the lifetime of a function to accommodate its complete state information. This information is used by the JVM to create a frame during the execution of a program. We first measure the reduction in frame size, realized by using VSS, during the compilation of a program. Fig. 13 shows that, for the programs in the last four categories in Table 2, the size of each frame is reduced by 29.9%, on average, for a slot size of 2 bytes and 30.10% for a slot size of 1 byte. Savings decrease when going from 2-byte to 1-byte slots because the benchmark programs made little or no use of 1-byte data types (i.e., `boolean` and `byte`).

We now measure the reduction in the RAM reserved for call stack during the execution of program. To this end, we record RAM reserved at each function call and at its return. Fig. 14

shows that at run time on average RAM reserved decreases by 11.16% and 10.03% with 2-byte and 1-byte slot size respectively. We note that the majority of benchmarks in Table 2 are taken from independent sources and hence are not designed specifically to save RAM using smaller than 4-byte data types. Thus, the reported savings may further increase if a programmer who is aware of the VSS optimization deliberately chooses smaller data types (e.g. `short` in place of `integer`) where appropriate.

In TakaTuka JVM, the RAM reserved with 1-byte slot size could be larger as compared to 2-byte slot size during program execution. This is because each slot uses one extra bit overhead and using 1-byte slot size usually double the number of slots, proportionally increasing that 1-bit overhead. If a program is using only a few 1-byte data types then the per-slot overhead may exceed RAM saved using 1-byte slot size as compared to 2-byte slot size.

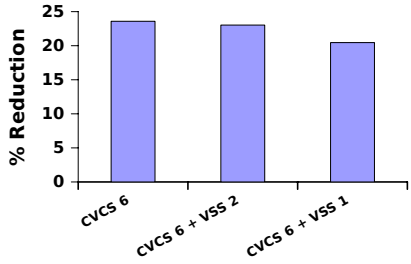
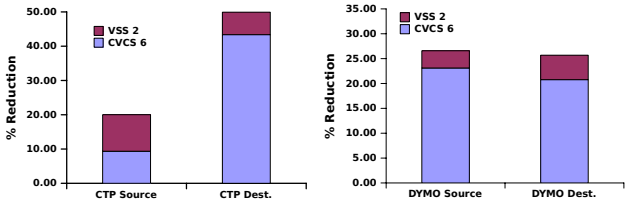


Figure 15. Execution Time: Average percentage reduction in the execution time of programs as compared to a frame based scheme.

4.2.3 Execution Speed

Fig. 15 shows that the benchmarks' execution time is reduced by 23.58% on average with CVCS. We attribute this reduction in execution time to the smaller number of memory operations and the reduction of FAT waste.

The IRIS mote has an 8-bit processor. Therefore, a program running on IRIS can access a memory address that is not a multiple of 32 bits without any overhead making IRIS suitable for VSS optimization. However, VSS engine increases the size of Java bytecode by adding casting instructions (as explained in Section 3), so that the number of instruction dispatches increases. These extra instruction dispatches result in slightly slower program execution with VSS. However, on average the programs run faster when both VSS and CVCS are used together as compared to the typical frame based scheme as shown in Fig. 15.

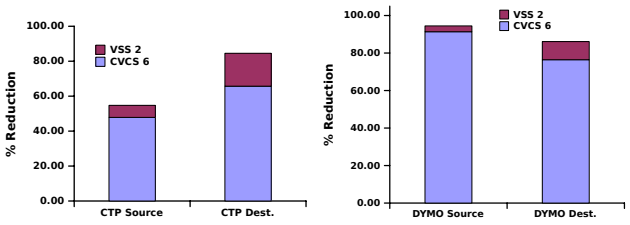


(a) CTP (b) Dymo

Figure 16. RAM Reserved in Routing Protocols: Average percentage reduction (relative to a frame-based scheme) in RAM reserved by the call stack with the use of CVCS and VSS.

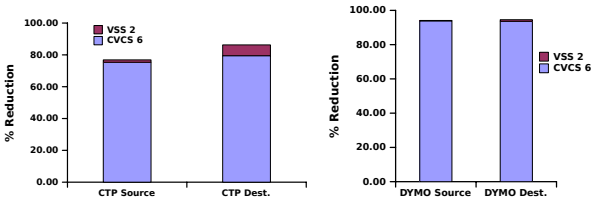
4.3 Multi-Machine Programs

We now evaluate CVCS and VSS using two well-known network layer protocols for wireless sensor networks (i.e. CTP and Dymo as



(a) CTP (b) Dymo

Figure 17. RAM Waste in Routing Protocols: Average percentage reduction (relative to a frame-based scheme) in RAM wasted by the call stack with the use of CVCS and VSS.



(a) CTP (b) Dymo

Figure 18. Number of Allocations in Routing Protocols: Average percentage reduction in the number of allocations by the CVCS (compared to a frame-based scheme).

listed in Table 2) . For evaluating CVCS/VSS with these protocols, we built a network of IRIS motes and designated two of the motes as a source and a destination.

The CTP is a tree-based collection protocol [11]. Few motes in a network advertise themselves as root of the network. Based on such advertisements, each non-root mote determines which of its neighbors is either a root or nearer to one of the roots. In steady state operation of CTP, a set of distributed trees are created by the non-root motes, each tree ending at a root mote. The CTP is an *address-free* protocol in which the data is always transmitted from a non-root towards the nearest root mote. In contrast, Dymo is a routing protocol where any mote can send packets to any other mote in the network [7].

We have developed a simple application on top of these multi-threaded network protocols. In our application, the source-mote sends a data packet (containing a counter) periodically towards the destination after a fixed interval of 500ms. The destination receives that packet through intermediate motes. The destination mote is connected with a desktop computer through a serial port. Each packet received at the destination mote is printed on the desktop computer attached to it.

The three metrics (RAM reserved, RAM waste and number of allocations) are measured before and after the invocation of each function. Fig. 16 shows that during program execution average RAM reserved is decreased by 19.13% and 49.92% at CTP source and destination, respectively, when both CVCS and VSS are employed. Similarly, RAM reserved for Dymo is decreased by 26.58% at the source node and 25.69% at destination with the use of CVCS and VSS. The corresponding reduction in the RAM waste for both network protocols is shown in Fig. 17. Finally, the number of allocations to maintain call stack is reduced by 88% on average for Dymo and CTP protocols (Fig. 18).

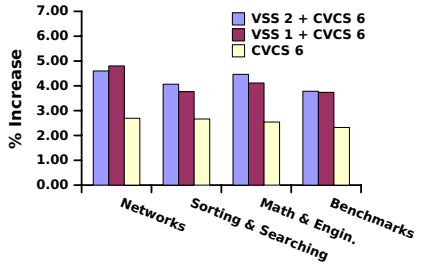


Figure 19. Percentage increase in the storage (flash) utilization.

4.4 Efficacy and Limitations

An IRIS mote has 128KB of flash memory and 8KB of RAM. TakaTuka is a small JVM that has to be stored in the mote's flash along with the Java binary of the program [3]. Our stack redesign reduces the amount of RAM required to run a program but it increases the size of the Java binary and of the JVM to be stored in flash memory. The JVM size is increased because new functions are added in the JVM to support CVCS and VSS. Furthermore, VSS adds new bytecode instructions to the Java binary to support smaller than 4-byte operations (as explained in Section 3). Fig. 19 shows that a program needs, on average, about 4% extra flash memory to store the modified JVM and the Java binary if VSS and CVCS are used together. This increase is not a problem because a mote is equipped with flash memory that is typically many times larger than the RAM [24]. Thus, we believe that this trade-off of a few KBs increase in flash storage to decrease the RAM is acceptable for wireless sensor motes.

5. Related Work

Much work in the area of embedded computing is geared towards reducing memory consumption. This survey of related work only considers approaches that concentrate on optimizing the handling of the run-time stack.

A groundbreaking work by Regehr and coworkers [17] applies abstract interpretation to ensure that there is no stack overflow. In contrast to our work, they consider interrupt-driven programs and do not change the frame layout (which is not an option because they are working with assembly language).

Choi and Han [8] consider optimizing for a hardware supported stacked register window, which could be regarded as a fixed size memory chunk managed by the processor. They also try to minimize the IFR waste by applying liveness analysis, but they do not consider the interaction with memory allocation in a multi-threaded run-time environment.

A similar topic is considered by Yang and coworkers [23]. They study an interprocedural algorithm that performs register allocation across procedures. Their goal is to manage the cost of the processor's window management.

Yi and coworkers [25] propose an adaptive scheme for adjusting the stack size in a sensor operating system based on a function's stack usage. This scheme is similar to our chunked allocation, but does not attempt to further optimize unused slots in stack frames. As another difference, they infer the frame sizes from assembly code, whereas our Java-based system obtains the frame size from the compiler.

Schäckeler and Shang [18] observe that many local variables are dead at recursive calls and propose to allocate them globally thus evading them from stack frames. Taken to the extreme, this idea amounts to the well-known tail call optimization [9], where the entire frame is reused by the next function call if all local variables are dead and there is no further computation after the call.

Many authors propose elaborate schemes for sharing or reallocating the stack space between different threads (e.g., [16]). However, these schemes operate at run time and do not touch the frame layout, whereas our work is based on compile-time analysis that optimizes the frame layout. Biswas and coworkers [5] propose a refined scheme, where (besides a run-time component) they permit a stack segment to grow into an area used by dead variables.

6. Conclusions

This paper presents a new design of the function call stack, using CVCS and VSS, to increase RAM availability and program execution speed.

CVCS allows a frame to overlap with other frames, thus reducing the RAM required for the execution of a program. By using chunks, CVCS also reduces the number of allocations required for maintaining the call stack. During compilation, it uses an inter-procedural data-flow analysis to estimate the size of memory chunks used at runtime for storing frames. During execution CVCS employs a shrinking strategy to reduce any overbooking of RAM caused by the limitations of compile-time analysis. To the best of the authors' knowledge such a compile-time stack redesign for object-oriented multi-threaded languages has never been attempted before.

VSS empowers the programmer to choose smaller than 4-byte slot size on an 8-bit or 16-bit wireless sensor mote during compilation. The selection of a smaller slot size saves RAM that would otherwise be wasted when a smaller than 4-byte data is stored in a 4-byte long slot. We have developed a data-flow analysis that changes the Java bytecode during compilation based on the selection of a slot size by the programmer.

We have used a wide variety of single and multi-threaded object-oriented programs, taken from independent sources to evaluate the performance of our schemes. As shown in Fig. 14 and Fig. 16, CVCS and VSS reduce the RAM required by the call stack of a program on average by 30%. They also reduce the number of allocations and deallocations by 80% (Fig. 12 and Fig. 18), on average.

In conclusion, our stack design enables the execution of feature-rich programs on a mote with tiny RAM with execution speed increased by 23% on average and increased battery lifetime. Our schemes could be adopted in any stack-based virtual machine with little or no modification.

Acknowledgements

We would like to acknowledge Iqbal Ali Azhar for his feedback on the paper. Furthermore, we are grateful to Safdar Iqbal and Abdur Rauf Rajar for finding the benchmarks to evaluate the performance of our call stack design.

References

[1] AlgoWiki: The Programmer's Compendium. http://algowiki.net/wiki/.

[2] Memsic: Wirless sensor networks. http://www.memsic.com/products/wireless-sensor-networks/.

[3] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, G. Ernst, E. Haussmann, S. Rührup, and Z. A. Uzmi. Optimized Java Binary and Virtual Machine for Tiny Motes. In *Distributed Computing in Sensor Systems (DCOSS)*, volume 6131, chapter 2, pages 15–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[4] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, and Z. A. Uzmi. Offline GC: trashing reachable objects on tiny devices. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 302–315, New York, NY, USA, 2011. ACM.

[5] S. Biswas, T. W. Carley, M. S. Simpson, B. Middha, and R. Barua. Memory overflow protection for embedded systems using run-time checks, reuse, and compression. *ACM Trans. Embedded Comput. Syst.*, 5(4):719–752, 2006.

[6] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a Feature-Rich VM for the Resource Poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 169–182, New York, NY, USA, 2009. ACM.

[7] I. Chakeres and C. Perkins. *Dynamic MANET On-demand (DYMO) Routing*. IETF (work in progress), 2010.

[8] Y. Choi and H. Han. Optimal register reassignment for register stack overflow minimization. *ACM Trans. Archit. Code Optim.*, 3(1):90–114, 2006.

[9] W. D. Clinger. Proper tail recursion and space efficiency. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 174–185, New York, NY, USA, 1998. ACM.

[10] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. volume 38, pages 1–11, New York, NY, USA, May 2003. ACM.

[11] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection Tree Protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 1–14. ACM, 2009.

[12] D. Gregg, M. A. Ertl, and A. Krall. A fast java interpreter. In *In Proceedings of the Workshop on Java*, 2001.

[13] Janice J. Heiss. Sentilla's Pervasive Computing – The Universe Is the Computer. 2008 JavaOne Conference.

[14] M. Johnson, M. Healy, P. Van De Ven, M. J. Hayes, J. Nelson, T. Newe, and E. Lewis. A comparative review of wireless sensor network mote technologies. *2009 IEEE Sensors*, pages 1439–1442, 2009.

[15] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[16] B. Middha, M. S. Simpson, and R. Barua. Mtss: Multitask stack sharing for embedded systems. *ACM Trans. Embedded Comput. Syst.*, 7(4), 2008.

[17] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embedded Comput. Syst.*, 4(4):751–778, 2005.

[18] S. Schäckeler and W. Shang. Stack size reduction of recursive programs. In T. Kim, P. Sainrat, S. S. Lumetta, and N. Navarro, editors, *CASES*, pages 48–52. ACM, 2007.

[19] M. Schoeberl, T. B. Preusser, and S. Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 120–127, New York, NY, USA, 2010. ACM.

[20] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4:2:1–2:36, January 2008.

[21] D. Spoonhower, G. Blelloch, and R. Harper. Using page residency to balance tradeoffs in tracing garbage collection. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 57–67, New York, NY, USA, 2005. ACM.

[22] Systronix. A practical engineering approach to using embedded java in real-world applications. http://www.systronix.com/book/benchmark/benchmark.html.

[23] L. Yang, S. Chan, G. R. Gao, R. Ju, G.-Y. Lueh, and Z. Zhang. Inter-procedural stacked register allocation for itanium like architecture. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 215–225, New York, NY, USA, 2003. ACM.

[24] X. Yang, N. Cooprider, and J. Regehr. Eliminating the call stack to save RAM. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '09, pages 60–69, New York, NY, USA, 2009. ACM.

[25] S. Yi, H. Min, S. Lee, Y. Kim, and I. Jeong. Sesame: space-efficient stack allocation mechanism for multi-threaded sensor operating systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1201–1202, New York, NY, USA, 2007. ACM.