# Optimized Java Binary and Virtual Machine for Tiny Motes

Faisal Aslam[1], Luminous Fennell[1], Christian Schindelhauer[1], Peter Thiemann[1], Gidon Ernst[1], Elmar Haussmann[1], Stefan Rührup[1], and Zastash A. Uzmi[2]

[1] University of Freiburg, Germany
{aslam,fennell,schindel,thiemann,ernst,haussmann,
ruehrup}@informatik.uni-freiburg.de
[2] Lahore University of Management Sciences, Pakistan
zartash@lums.edu.pk

**Abstract.** We have developed TakaTuka, a Java Virtual Machine optimized for tiny embedded devices such as wireless sensor motes. TakaTuka[1] requires very little memory and processing power from the host device. This has been verified by successfully running TakaTuka on four different mote platforms. The focus of this paper is TakaTuka's optimization of program memory usage. In addition, it also gives an overview of TakaTuka's linkage with TinyOS and power management. TakaTuka optimizes storage requirements for the Java classfiles as well as for the JVM interpreter, both of which are expected to be stored on the embedded devices. These optimizations are performed on the desktop computer during the linking phase, before transferring the Java binary and the corresponding JVM interpreter onto a mote and thus without burdening its memory or computation resources. We have compared TakaTuka with the Sentilla, Darjeeling and Squawk JVMs.

## 1 Introduction

A common way of programming an application for wireless sensor motes is by using a low level programming language such as Assembly, C and NesC [3]. These languages tend to have a steep learning curve and the resulting programs are difficult to debug and maintain. In contrast, it is attractive to program these motes in Java, a widely used high level programming language with a large developer community. Java is highly portable and provides many high level concepts including object oriented design, type safety, exception handling and runtime garbage collection. However, Java portability requires a virtual machine, which comes with significant memory and computation overhead [7]. Therefore, it is difficult to run such a virtual machine on a wireless sensor mote which typically has a 16 or an 8 bit microcontroller with around 10KB of RAM and 100KB of flash [18]. Furthermore, these motes cannot perform computation intensive tasks due their limited battery lifetime. Therefore, a Java Virtual Machine (JVM) designed for such tiny embedded devices must have low computation requirements,

[1] The complete source of TakaTuka is available at `http://takatuka.sourceforge.net`

R. Rajaraman et al. (Eds.): DCOSS 2010, LNCS 6131, pp. 15–30, 2010.

a small RAM footprint, and small storage requirements. To satisfy these stringent requirements, we have designed TakaTuka, a JVM for wireless sensor motes.

The focus of this paper is TakaTuka's optimization for reduction in flash storage requirements of the Java classfiles and virtual machine. The main contributions described in this paper are: 1) Extensive Java bytecode and constant pool optimizations. As part of these, we present a novel *optimal* bytecode replacement algorithm (Section 3.3.2). 2) A condensed Java binary format (called Tuk). Beside providing size reduction, the Tuk file format reduces RAM and computation requirements by enabling constant time access to preloaded information stored in flash. 3) A novel design for dynamically sizing the JVM interpreter. 4) TakaTuka Java interfaces for typical mote hardware and an implementation based on TinyOS. The current version of TakaTuka can supports *all* mote platforms using TinyOS that either have MSP430 or AVR family processors. We have acquired four of those platforms and successfully tested TakaTuka on them. This includes Crossbow's Mica2, Micaz, TelosB and Sentilla's JCreate [18] [17].

**Outline:** We present related work and background information in Section 2. Java bytecode optimization is described in Section 3 and constant pool optimization in Section 4. The Tuk file format, is discussed in, Section 5 and our JVM design is presented in Section 6. We discuss TakaTuka's linkage with TinyOS in Section 7. We present results in Section 8 and finally, in Section 9, we draw the conclusions.

## 2 Related Work and Background

This section summarizes the existing JVMs for tiny embedded devices and bytecode size reduction techniques.

**JVM for Motes:** Recently, Sun Microsystems has developed *Squawk*, a JVM for embedded systems [7], which overcomes some of the shortcomings of traditional JVMs by employing a *Split VM Architecture* (SVA). In SVA, resource-hungry tasks of the JVM, including class file loading and bytecode verification are performed on the desktop [14] [7]. This process reduces the memory and CPU usage requirements for the execution of the program on the mote, because no runtime loading and verification is required. When compared to standard JVMs, Squawk has less stringent requirements for resources; it is, however, still not feasible to run Squawk on a typical mote equipped with an 8-bit microcontroller, a few hundred KB of flash and around 10KB of RAM [7]. For these typical motes, Sentilla Corp. has developed a JVM, but it is not open-source and currently does not support any devices other than Sentilla motes [17]. Darjeeling, is an open source JVM designed for motes [11]. It does support a good part of the JVM specification but sacrifices some features like floating point support, 8-byte data types and synchronized static-method calls for efficiency [11]. There are a few other JVMs available for embedded devices, such as NanoVM [1], and VM* [10], but these are either limited in functionality by not fully supporting JVM specifications or are closed source with a limited scope of operation only on specific devices. TakaTuka aims to remain small, open source and suitable for a wide variety of embedded devices while providing all features of

a *fully CLDC-compliant JVM*[2]. The current version of TakaTuka supports all but two of the Java bytecode instructions and most of the CLDC library. We also support threading, synchronized method calls, 8-byte data types and 4-byte floating point arithmetic on motes.

**Bytecode Optimization:** The two primary methods for bytecode size reduction are *compression* and *compaction* [6][5]. A typical compression technique requires partial or full decompression at runtime [6]. Any decompression always results in computation and memory overhead. Therefore, performing decompression at runtime is not desirable for embedded devices. In contrast to compression, *compaction* involves identifying and factoring out recurring instruction sequences and creating new customized bytecode instructions to replace those sequences [2]. These customized instructions do not require decompression and are interpretable because they share the characteristics of normal bytecode instructions. The process of compaction produces a smaller code that is executable or interpretable with no or little overhead. The compaction scheme given in [2] is shown to produce bytecode that is about 15% smaller and runs 2 to 30% slower than the original bytecode. Rayside et al. [5] compaction scheme produces bytecode that is about 20% smaller. In contrast to above mentioned compaction approaches, TakaTuka comprehensive compaction scheme produces a bytecode reduction of about 57% on average and the resultant bytecode runs faster without using any extra RAM.

## 3 TakaTuka Bytecode Compaction

We employ three bytecode compaction techniques, each of which replaces a single bytecode instruction or a sequence of bytecode instructions with a new *customized bytecode instruction* such that the total size of the bytecode is reduced. A customized bytecode instruction, like any other bytecode instruction, is composed of an opcode and an optional set of operands. In the following, we first explain the process of choosing an opcode for a customized instruction. Then, we provide the details of the compaction processes and relevant algorithms used in TakaTuka.

### 3.1 Available Opcodes

Each customized instruction uses an opcode that is not used by any other bytecode instruction. Hence the cardinality of the set of available opcodes impacts the extent of compaction. The Java specification has reserved one byte to represent 256 possible Java opcodes but uses only 204 of those for corresponding bytecode instructions [14]. Thus, there are 52 unused opcodes that are available for defining customized instructions. Furthermore, the Java specification includes many bytecode instructions with similar functionalities but different data-type information. The type information of such instructions is only used

[2] We might never actually obtain formal CLDC-compliance due to the high price tag associated with the license of the CLDC Technology Compatibility Kit (CLDC TCK).

in Java bytecode verification and is not required by the JVM interpreter. Since the Split VM architecture (SVA) does not require run-time verification (see Section 2) additional 29 opcodes for a compaction algorithm are available after completing bytecode verification during linking phase. Finally, many Java programs may not use all the 204 standard Java instructions, depending upon the functionality of the program. Hence a custom-made JVM interpreter such as the one offered by TakaTuka can make use of additional opcodes, not used by the Java program, for the purpose of defining customized instructions during the bytecode compaction process.

### 3.2 Single Instruction Compaction

In this technique of compaction, size of a single bytecode instruction is reduced by replacing it with a smaller customized instruction. That is, in single instruction compaction, each customized instruction replaces only a single bytecode instruction. The single instruction compaction in TakaTuka can either be a reduction in the memory footprint needed to represent an operand (called *Operand reduction*) or a complete removal of the Operand from the bytecode instruction (called *Operand removal*).

**Operand reduction:** Many instructions in standard Java use either a 2-byte constant pool (CP) index or a 2-byte operand as a branch offset [14]. In TakaTuka, we introduce a new custom instruction with a reduced operand size of one byte, if the operand value is smaller than 256. In order to maximize the savings resulting from the use of this technique, we sort the information in our set of global CPs[3] such that most referred entries of a CP from the bytecode are stored at a numerically small CP index. This leads to a large number of reduced size constant pool instructions in the bytecode.

**Operand removal:** In TakaTuka, we also combine the opcode and operand to form a customized instruction with implicit operand(s). For example, the instruction `ILOAD 0x0010` could be converted to `ILOAD_0x0010` and the two bytes originally used by the operand may be saved. Note, however, that we do not apply operand removal on offset-instructions as their offset usually changes after any kind of bytecode compaction.

### 3.3 Multiple Instruction Compaction (MIC)

In multiple instruction compaction (MIC), a recurring sequence of instructions, called a *pattern*, is replaced by a single *customized instruction*. For example, a pattern {`GOTO 0x0030, LDC 0x01, POP`} in bytecode could be replaced by a single customized instruction `GOTO_LDC_POP 0x003001`, providing a reduction of two bytes per occurrence. Note that, the MIC technique perform compaction in the opcodes only, without affecting the storage needed for operands. The MIC

[3] Global constant pool is explained in detail in Section 4.

technique involves finding a set of valid patterns and then replacing a subset of those patterns with customized instructions. First, we define the criteria that must be met for a pattern to be replaceable and then we discuss the valid pattern search and replacement algorithms used in TakaTuka.

**Valid Pattern Criterion:** A sequence of bytecode instructions or a pattern is said to be *valid* if it can be replaced by a single customized instruction. A valid pattern fulfils the following two criterion: 1) A branch-target instruction can only be the first instruction of a pattern, and 2) Any Java bytecode instruction designed to invoke a method can only be a last instruction of a pattern. Note that above restrictions are imposed to avoid extra computation or RAM required for decoding a customized instruction during runtime for finding a return or branch offset target inside it.

### 3.3.1 Pattern Identification

The pattern identification algorithm finds and selects a number of patterns of instruction sequences from the original bytecode of the Java program, up to a maximum number of available opcodes. These patterns are stored in a hash-map which is used as an input to the pattern replacement algorithm. The pattern replacement algorithm then constructs customized instructions and replaces the input patterns with those customized instructions in the bytecode. We use the following terminology to explain the pattern identification and replacement algorithms.

- $m$ : Total number of opcodes that may be used by a customized instruction.
- $k$ : Maximum number of single instructions in any pattern.
- $l_i$ : Number of single instructions in a pattern $i$ that can potentially be replaced by a customized instruction. We also refer to this parameter as the length of the pattern $i$.
- $\epsilon_i$ : Reduction in bytecode achieved when one occurrence of a pattern $i$ in the bytecode is replaced by a customized instruction. $\epsilon_i$ equals $l_i - 1$ when a pattern $i$ is replaced by a MIC customized instruction.
- $\zeta_i$ : Frequency of a pattern $i$, that is to be replaced by a customized instruction, in the entire bytecode of the Java program.
- $\eta_i$ : Total reduction (i.e. $\epsilon_i \cdot \zeta_i$) in bytecode achieved when a pattern $i$ is replaced by a customized instruction, in the entire bytecode of the Java program.
- $\xi_{(y)} : \sum_{i \in y} \eta_i$ where $y$ is a set of patterns.

In TakaTuka, pattern generation for multiple instruction compaction uses a Multi-pass greedy algorithm, which is based on a simple Single-pass greedy algorithm.

**Single-pass greedy algorithm:** The Single-pass greedy algorithm creates a list of patterns of length $\leq k$ by traversing the bytecode exactly once. When a valid pattern $i$ of any length is encountered the first time, it is added to the hash map with $\zeta_i = 1$. Then, $\zeta_i$ is incremented whenever the same pattern $i$

is found again while traversing the remaining bytecode. Consequently, after a single traversal of the Java bytecode, the hash map contains all possible patterns of length $\leq k$ with their corresponding frequencies. The algorithm returns a subset $\sigma$ of patterns from within the hash map such that $|\sigma| \leq m$ and $\xi_{(\sigma)}$ is maximized. This algorithm has one major flaw: it returns many patterns that are not new pattern but subset of other longer patterns, undermining the extent of bytecode reduction.

**Multi-pass greedy algorithm:** The multi-pass greedy algorithm mitigates the limitation described above by traversing the bytecode multiple times, making temporary changes in each iteration, and using that changed bytecode in subsequent iterations. In the first iteration, the single-pass greedy algorithm is used on a copy of the bytecode and the resulting patterns are stored in a set $y$. A pattern $i$ is then selected, such that $\eta_i \geq \eta_j \ \forall j \in y$, and replaced as a customized instruction in the copy of the bytecode. Subsequent iterations are similar except that the single-pass greedy algorithm is called on the modified copy of bytecode from the previous iteration. This continues until either $m$ patterns of length $\leq k$ are selected or additional patterns cannot be found. The customized instructions introduced in a given iteration may become a part of a new pattern in subsequent iterations, as long as the constraint of maximum $k$ original single instructions per customized instruction is not violated.

### 3.3.2 Pattern Replacement

The pattern replacement algorithm takes the bytecode and a set of patterns and replaces those patterns, as they appear in the byteode, by new customized instructions. The primary goal in this replacement process is to maximize the bytecode reduction, leading to the maximum savings in storage. While our pattern generation algorithm is greedy and may not generate an optimal set of patterns, our pattern replacement algorithm is not only *optimal* but also runs in polynomial time. First, we describe our algorithm, then we show its polynomial complexity in Theorem 1, finally proving its optimality in Theorem 2.

**Algorithm:** The pattern replacement algorithm keeps track of many temporary solutions in order to produce the replacement with the maximum savings. The algorithm is applied on each class method within the bytecode one by one and produces the maximum reduction possible for that method with the given set of pattern. The inputs to the pattern replacement algorithm are: 1) the number $k$ indicating the maximum number of single instructions in any pattern, 2) a set of patterns $\sigma$ generated by a pattern identification algorithm and 3) the bytecode of a method. The pattern replacement algorithm creates a tree with different replacement possibilities. One branch of this tree contains the bytecode sequence corresponding to the maximum reduction in bytecode, and the algorithm uses this branch to update the bytecode of the method. To demonstrate the replacement algorithm, assume that the instruction at index $i$ in the bytecode of a method $\mu$ is represented by $\tau_i$. That is, $\{\tau_1, \tau_2, ..., \tau_\lambda\}$ represents the method bytecode, where $\lambda$ is the total number of instructions in $\mu$. Each level in the tree corresponds to the index in the bytecode, hence the tree has depth

$\lambda$. Within the tree, each node located at level $j$ corresponds to either $\tau_j$, or to a customized instruction that ends at $\tau_j$. Each node $x$ in the tree has exactly one incoming edge whose weight $w(x)$ is given by:

$$w(x) = \begin{cases} 1 & x \text{ is the root node} \\ 0 & x \text{ corresponds to a customized instruction,} \\ \min(k-1, w(x_p)+1) & x \text{ corresponds to } \tau_j \text{ itself} \end{cases} \quad (1)$$

where $x_p$ is the immediate parent of node $x$. Note that each node $x$, other than root node, has exactly one immediate parent node. Each node $x$ in the tree has at most $w(x)+1$ child nodes each corresponding to instructions with unique lengths ranging from 1 to $w(x)+1$. If node $x$ exists at a level $j$ in the tree, then one of its child nodes corresponds to the instruction $\tau_{j+1}$ which has length 1; each of the other $w(x)$ child nodes corresponds to a customized instruction $c$ that represents a pattern obtained by traversing node $x$ and parents further above in the tree, such that $2 \leq l_c \leq w(x)+1$. Each node of the tree maintains the total bytecode reduction achieved by the tree branch ending at it.

The tree is built level by level, where addition of each level is done in two phases: the *creation phase* and the *pruning phase*. The creation phase of a level $j$ is carried out simply by finding the children of all the nodes at level $j-1$. In the pruning phase of level $j$, first we prune all those nodes from level $j$ which represent an invalid customized instruction i.e. its corresponding pattern is not a member of $\sigma$. Subsequently, additional nodes are pruned such that no two nodes have the same weight on their incoming edge: if multiple nodes have $w$ as the weight on their incoming edge, the one corresponding to the leaf of the branch with highest total bytecode reduction is kept and the remaining nodes are pruned. In this additional pruning, random selection is made if there is a tie.

Each level of the final tree has at most $k$ nodes because the weight allowed on any incoming edge is between 0 and k-1 (see Equation 1) and after pruning nodes on each level have edges with distinct weights. Thus, the resulting tree is a linear sized structure with depth $\lambda$ and a constant span $k$. After the tree with level $\lambda$ is completely constructed, the leaf node with the highest saving is identified and the corresponding tree branch is used to replace the bytecode.

**Theorem 1.** ***The time complexity of the replacement algorithm is*** $O(k^2 \cdot \lambda)$.

*Proof.* The complexity of the algorithm depends on the size of the tree and the number of operations performed on each node. From the description of the algorithm, we note that the depth of the tree is exactly $\lambda$ and each level of the tree has at most $k$ nodes. The final tree has at most $k \cdot \lambda$ nodes. However, in the worst case, before a pruning phase, a level $j$ may have a total of up to $1+2+3+...+k$ nodes. Pruning nodes at level $j$ means finding the node with the maximum reduction for each weight $0 \leq w \leq k-1$. This can be done with

an effort proportional to the number of nodes at this level, i.e. $\frac{1}{2}k(k+1)$ nodes in the worst case, which is bounded by $O(k^2)$. Since there are exactly $\lambda$ levels in the tree, the worst case complexity of the algorithm is $O(k^2 \cdot \lambda)$.

**Lemma 1.** ***All nodes with same weight*** $w$ ***on their edge at level*** $j$ ***will have same sub-tree originating from them.***

*Proof.* We prove our claim by contradiction: assume that all nodes with the same weight $w$ on their incoming edges at level $j$ do not have the same sub-tree originating from them. This is possible if and only if their immediate children are different. We now consider two situations: (i) when $w$ is zero, each node has only one child node corresponding to the instruction $\tau_{j+1}$. Thus, all sub-trees are the same, contradicting the original assumption, and proving the lemma. (ii) when $w$ is non-zero, a child node either has a customized instruction composed of at most $w$ parent nodes or a simple instruction $\tau_{j+1}$. Based on Equation (1), all of those $w$ parents of each node will always be simple instructions instead of customized instructions. The algorithm says that a simple instruction $\tau_i$ can only occur at level $i$ in tree. Hence all of those $w$ parents nodes must be equal. In summary, the immediate children of each node with same $w$ at level $j$ will have the same $w$ parents and, therefore, will have the same child nodes. Hence the sub-tree emanating from each node with same weight $w$ at level $j$ will always be the same.

**Theorem 2.** ***Given a set of patterns, the pattern replacement algorithm finds the replacement with maximum overall reduction in the size of bytecode.***

*Proof.* First, we note that the complete tree without pruning contains all combinations of solutions including the optimal one identified by the leaf node with the highest total reduction. Next, we argue that the branch corresponding to the optimal solution is not affected by pruning. Using Lemma 1, all nodes with the same weight $w$ on their incoming edge at level $j$ have the same sub-tree emanating from them. Therefore, pruning all nodes with the same weight $w$ except the one with maximum saving achieved so far, implies that the optimal solution is still part of the tree.

## 4 TakaTuka Constant Pool Optimization

Each class file has a collection of distinct constant values of variable size called the *Constant Pool* (CP). These constant values either define the characteristics of the class or are used by the class itself. A two byte index is used to access a given CP value which is usually larger than two bytes and is used multiple times from the class file. The aggregated CP size of a project is usually much larger compared to its total bytecode size. Hence, reducing CP size is critical in the overall size reduction of a Java program. Our constant pool design is based on some of the ideas given in [5] and [13] with improvements drawn using the

characteristics of a Split-VM architecture, which is not considered in above references. In the following sub-sections we present the optimizations we used in TakaTuka for reducing the CP size.

**Global Constant Pool:** Each value of a CP entry could be of one of the eleven different types as specified in the Java specification [14]. In traditional designs, the CP values of a single class appear in an arbitrary order within the CP, where a leading one byte tag is used for type identification. This design, however, has the following shortcomings: 1) One byte is consumed to specify the type with each CP entry. 2) Since a CP is unordered, an index has to be built in RAM in order to index its entities in a constant-time. 3) Although CP values are distinct for one class, there can be many redundant occurrences in the scope of a given project.

The above mentioned shortcomings lead to excessive flash and RAM requirement, both of which are scarce resources in sensor motes. To address this in TakaTuka, we use the preloading characteristic of SVA and create one *global* pool per type, during the linking phase. As compared to traditional CPs, our set of Global Constant Pools (GCPs) have no redundant information per project. We keep a common header for these GCPs specifying the start address of a pool and corresponding type. As all entries of a GCP have the same type, no tag is required per constant pool entry. Keeping a separate CP per type enables a constant time lookup for any CP's entry in the flash and does not require loading the complete CP in RAM. This is because each CP type has only fixed size values[4] hence given a constant pool type one can directly jump to a specify CP index by computing the offset from the first entry of the same type. This constant time lookup is possible because each Java CP-instruction always accesses the same type of CP (e.g., `INVOKEVIRTUAL` always access CP of type 10). However, there are three exceptions to this namely the instructions `LDC`, `LDC_W` and `LDC2_W`. We have introduced five additional bytecode instructions so that in TakaTuka each CP instruction, including the ones mentioned above, implicitly contain the CP type information.

**Reference Resolution:** Traditional JVMs apply dynamic loading, also called on-demand loading. Whenever a class method or a field needs to be accessed, the corresponding class file has to be loaded into RAM after performing verification, preparation and resolution [14]. To resolve references during runtime, fully qualified names are required to identify components (i.e. methods, fields and, classes). In TakaTuka, we have used the preloading characteristic of SVA to resolve names during linking. Hence a preloaded, preverified and resolved Java program is transferred to a mote. This allows us to remove all the UTF-8 strings traditionally required for name resolution but not used by the application. Furthermore, we can also remove all the other constant pool entries (e.g., all entries of type 12) typically used for resolving names during runtime[14].

[4] UTF-8 constant pool type has variable length values but they are never directly used from within the bytecode [14].

## 5 Tuk File Format

The classfile has three important parts: the bytecode, the CP and the structure information. We observed that the structure information usually makes up for half the size of classfiles of a program. We use a special format called *Tuk* for storing the set of classfiles used by a user-program and corresponding Java library components. The Tuk format has two main characteristics. First, it only contains information required during program execution in a reduced format, forgoing any information required during linking phases. Second, the Tuk format contains preloaded information stored in an easy to access manner, obviating the need to load the Tuk file in a mote's RAM. This implies that different portions of a Tuk file can be accessed in constant time using pre-calculated addresses, relieving the computation resources of the host device. It may be noted that the addresses and indexes that make up part of the preloaded information are created by processing on a desktop computer, before transferring a small preloaded and preverified Tuk file to the host device. For example, in a Tuk file a CP entry for a class or a method reference contains the address of the location inside the Tuk file where actual data (i.e., the classfile or the method) resides. All the addresses in a Tuk file are relative to its starting address and are therefore platform independent. We use either 2-byte or 4-byte addresses depending upon the total size of the Tuk file.

## 6 TakaTuka JVM Design

In this section, we present the design of TakaTuka's dynamically customizable JVM interpreter, and its bytecode compaction support.

**Customized JVM:** The flash memory in some tiny embedded devices may be too small to contain a complete JVM. To address this limitation, the default behavior of TakaTuka is to reduce the size of the interpreter depending upon the set of bytecode instructions used by a given Java program. To this end, TakaTuka removes all unused components from the JVM, stripping the JVM bytecode support down to the bytecode set used by the given program. For a given Java program, the initial step of compilation is to generate a *header file* enlisting the set of bytecode instructions used by that program. This step is completed on the desktop computer. Subsequently, the JVM recompiles to shrink itself based on the information contained in the header file. This default behavior leads to a very small JVM interpreter, albeit the one that is capable of running only a subset of Java programs. If a more generic interpreter, capable of supporting additional bytecode instructions, is needed, TakaTuka allows this through a configuration file. For example, it is possible to have a JVM interpreter version that supports the complete set of bytecode instructions except the ones which involve floating point operations. Similarly, a user can also completely turn off the JVM customization, resulting in the generation of a general purpose JVM that supports any CLDC-compliant Java program.

**Bytecode Compaction Support:** The TakaTuka interpreter is implemented by using the *labels-as-values* approach [4] to provide direct threading. Each bytecode instruction translates into a source code snippet following a corresponding *label*. To this end, TakaTuka treats the customized instructions, resulting from single instruction compaction and multiple instruction compaction, like any other bytecode instruction that could not be compacted. However, because some instructions are only generated during the compaction process a 'static' set of labels is not applicable. TakaTuka addresses this problem by dynamically generating the labels supporting the required set of bytecode instructions. In case a user wishes to generate a general purpose JVM interpreter that can run any CLDC-compliant program then a set of *fixed* customized instructions is used, instead of generating new customized instruction per application.

## 7 TakaTuka Linkage with TinyOS

TakaTuka can run on all devices currently supported by TinyOS. Furthermore, TakaTuka provides a Java interface for each driver and a user can access the class implementing such an interface using only a factory-method [8], making the actual implementation hidden from the users of the driver. Therefore, TakaTuka can run on any platform that provides an implementation of those Java interfaces using any operating system, including but not limited to TinyOS [5].

To work with TinyOS, the TakaTuka interpreter is allowed to execute $n$ bytecode instructions in a scheduling cycle before returning the control back to the TinyOS scheduler. This is accomplished by posting a TinyOS Task [12] that calls the TakaTuka interpreter which is written in C. We keep $n$ small so that TinyOS remains responsive in between TakaTuka scheduling cycles. TinyOS is an event driven operating system. In contrast Java language functions are generally blocking and it is up to Java user to create threads when multitasking is required. Hence when a TakaTuka user calls a method that is written as event in TinyOS then that method blocks until the corresponding event is generated in TinyOS. When TinyOS receives an event, then the Java thread waiting for that event is notified. A TakaTuka user can access any of the TinyOS Commands [12] using TakaTuka native method support. For a Split-phase TinyOS Command [12], the current thread is put into a waiting state after calling the Command and is notified when latter is completed. TakaTuka also supports thread synchronization, that is used for sharing resources (e.g. radio) between multiple user threads.

**Power Management:** The current version of the TakaTuka JVM uses the same power management strategies as implemented in TinyOS but in future versions, we intend to provide a more comprehensive power management. In the current version, when all Java threads are sleeping or waiting for events then TinyOS could go to the low power state and TakaTuka follows through. In case an event arrives, then any thread waiting for that event is notified and and TakaTuka resumes. Furthermore, TakaTuka uses TinyOS radio drivers, thereby supporting the low-power listening implemented by TinyOS.

[5] The current version of TakaTuka has also limited support for Contiki OS.

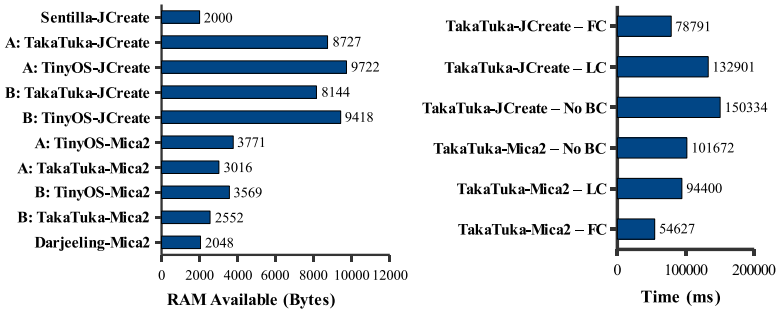


**Fig. 1. Left-Fig:** RAM available (in bytes) for a user program. Unlike the Sentilla and Darjeeling JVMs, the RAM available for a user program in TakaTuka and TinyOS depends on drivers used by it. We have presented two kinds of results for TinyOS and TakaTuka. a) When only serial drivers are used and b) When serial, radio and LED drivers are used. **Right-Fig:** Time (in ms) required to sort an array of 1500 bytes using the Quicksort algorithm. TakaTuka results are shown when no Bytecode Compaction (no BC), Full Compaction (FC) and Limited Compaction (LC) is used.

## 8 Discussion and Results

This section presents TakaTuka storage results as compared to the Darjeeling, Sentilla and Squawk JVMs.

### 8.1 Execution Speed and RAM

We show that our storage optimization does not have an adverse effect on RAM usage and performance. In contrast the bytecode optimization results in enhanced performance.

**RAM available for a user program:** The TakaTuka interpreter and the low level drivers require a few hundred bytes of statically allocated data memory. The exact amount depends on the used functionality. Furthermore we assume a worst case upper bound of 500 bytes for the stack memory required by the interpreter's routines. The remaining RAM is available as Java heap and stack memory for the user program and the runtime library. It is shown in Fig. 1, that TakaTuka has 81.44%, TinyOS has 94.18% and Sentilla JVM has 20% of the total RAM available for a program that uses serial, radio and LED drivers on a JCreate mote. For the Mica2 mote Takatuka has 62.30%, TinyOS has 87.13%, and Darjeeling has 50% of the total RAM available for the user program when LED, radio and serial drivers are used. The Darjeeling JVM does not adjust the amount of RAM available automatically. It can be set at compile time and defaults to 2048 bytes for Mica2.

**Execution speed**: The increase of execution speed when using bytecode compaction is shown in Fig. 1. On JCreate, the TakaTuka JVM runs 47.59% faster with Full Compaction and 11.6% faster with Limited Compaction as compared to when no compaction is used. Bytecode compaction results in a similar speed increase on Mica2.

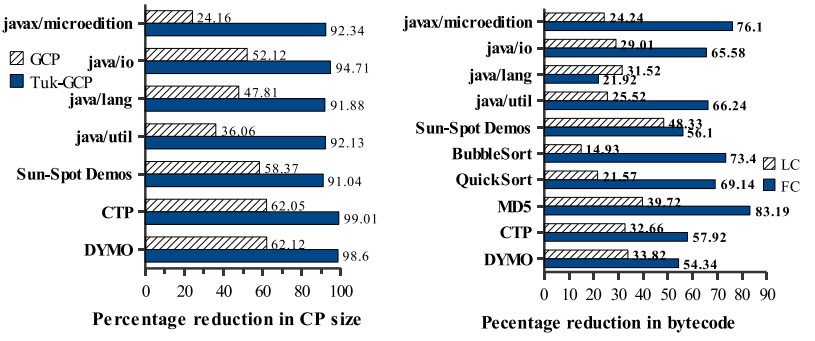


**Fig. 2. Left-Fig.** Percentage reduction in GCP and Tuk-GCP sizes as compared to original CP sizes of classfiles. **Right-Fig.** Percentage reduction in bytecode using full compaction (FC) and limited compaction (LC).

### 8.2 Storage

In this section we present TakaTuka storage optimization results.

**Input applications:** In addition to simpler well known algorithms (Quicksort, Bubblesort, MD5) we have used DYMO and collection tree (CTP) routing protocols as input to our optimization algorithms[9][16]. Both Dymo and CTP are completely written in Java using multiple threads. Our input files to the TakaTuka optimization framework for above mentioned applications include user defined classfiles as well as all other linked library files used by the user program. We have also used CLDC library packages and Sun-Spot Demos to produce storage results. However, in this case, the input files do not include external packages and no dead-code removal is applied.

**CP optimizations:** We first create a set of 11 Global Constant Pools (GCPs) with no redundant information for each application, subsequently the reference resolution information is removed (Tuk-GCPs). On average the aggregated size of GCPs are 48.96% and the Tuk-GCPs are 94.24% smaller then the original corresponding classfiles aggregated CPs (Fig. 2).

**BC optimizations:** Our bytecode optimization not only reduces the overall Java binary size but also increases execution speed, as shown in the previous Section 8.1. In case a customized TakaTuka JVM is used, the bytecode compaction results in regenerating the interpreter's labels for new customized instructions (Section 6). Hence we define the net reduction in the Java binary size due to bytecode compaction for a given pattern $x$ by:

$$R(x) = \eta_x - \gamma_x \tag{2}$$

where $\eta$ is defined in Section 3.3.1 and used here in the context of all three compaction techniques. $\gamma_x$ is the total change (increase or decrease) in the size of the label's block for supporting a customized instruction that is replacing a given pattern $x$ of original Java instructions. Through configuration, we allow two compaction strategies in TakaTuka: Limited Compaction (LC) and Full

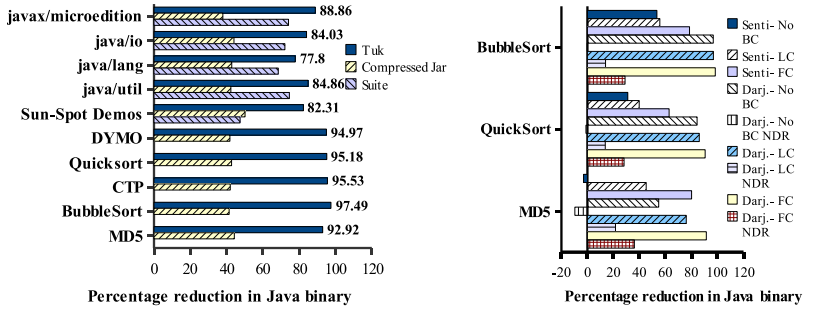


**Fig. 3. Left-Fig.** Overall reduction in Java binary for TakaTuka applications, SunSpot demos and CLDC libraries when LC is used. The applications include user and corresponding library code. In contrast, the Sun-Spot demos as well as the CLDC libraries do not include classfiles from external packages. **Right-Fig.** Overall reduction in Java binary for TakaTuka compared to the Sentilla (Senti) and Darjeeling (Darj) JVMs. The different configurations for TakaTuka are: without bytecode compaction (No BC), with Limited Compaction (LC), with Full Compaction (FC), and without dead-code removal (NDR).

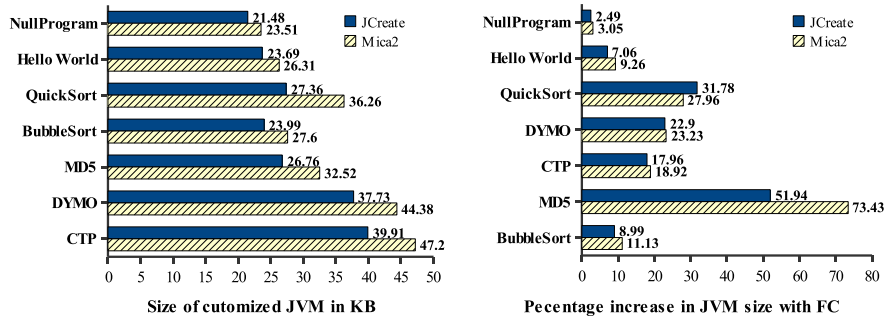


**Fig. 4. Left-Fig.** Overall size (in bytes) of the customized TakaTuka JVM for different applications. **Right-Fig.** The percentage increase in customized TakaTuka JVM size when Full Compaction (FC) is used.

Compaction (FC). In LC the change in size of the label's block is taken into account and compaction is only performed when the compaction reduction function of Equation 2 returns a positive value. In contrast, in FC any change in the size of the label's block is ignored and compaction is always performed if unused opcodes are available. As shown in Fig. 2, on average FC reduces the size of the Java bytecode by 62.39% and LC by 30.14%. By default we use LC instead of FC on JCreate as the net size of the Java binary and the JVM interpreter is increased when FC is used as shown in Fig 4. On Mica2 we can use FC to profit from the performance increase as its bigger flash memory can tolerate the storage increase more easily.

**Overall Java binary reduction:** It is shown in Fig. 3, that for the given programs, TakaTuka's Java binary file, called Tuk, is on average 24.12% smaller compared to corresponding Suite files. Furthermore the Tuk file size is reduced by 95.22% for applications and 83.57% for libraries compared to corresponding uncompressed Jar files. The right part of Fig. 3 compares the size of Tuk files resulting from different compaction configurations with the Java binaries produced by Sentilla and Darjeeling. In the comparisons with Sentilla only the user application is considered, as the library is already present on the Sentilla motes and generally not recompiled. In the comparisons with Darjeeling we also compare the size of the Tuk file without dead-code removal, as the Darjeeling JVM do not perform dead-code removal. TakaTuka always shows significant reduction in the size of the Java binary when using bytecode compaction.

**Customized JVM reduction:** As discussed in Section 6, TakaTuka supports both customized JVM per user program as well as general purpose JVM that can run any CLDC program. The overall size of the general purpose JVM for Mica2 is 56914 bytes and 34538 bytes for JCreate mote. Hence a customized JVM for the `Null Program` is 58.70% and 37.80% smaller compared to the general purpose JVM on a Mica2 and JCreate respectively, as shown in Fig. 4. The figure also shows customized JVM sizes for other programs. Darjeeling produces a general purpose JVM of 60800 bytes.

## 9 Conclusion

We have developed TakaTuka, a JVM that runs on platforms with RAM as small as 4KB and flash as small as 48KB. Using TakaTuka, we also developed a number of Java WSN applications and verified that their storage requirements are kept small on tiny platforms without increasing CPU and RAM usage. TakaTuka takes several measures to curtail the storage requirements, that include: 1) A condensed format for classfiles called Tuk. 2) Comprehensive CP reduction techniques. 3) Various bytecode reduction strategies including single and multiple bytecode compaction and a novel optimal bytecode replacement algorithm. 4) A JVM that may be customized for a user application.

These measures have resulted in an average size reduction of 95.22% for the resulting Tuk files, given a set of user Java applications, as compared to the original classfiles. Furthermore, the customized TakaTuka JVM size is shown to reduce up to 58.70%.

TakaTuka bytecode compaction results in a performance gain of 47.59% on average. This is because the customized instructions reduce the overall time needed for instruction dispatch and fetching instruction operands, which are the primary time-consuming tasks of typical JVM interpreters [15]. Furthermore, each entry in a global CP or any other part of Tuk file is accessable in a constant time, leading to reduced computation and RAM requirements at run-time.

In summary, our results for a variety of applications on a multitude of hardware platforms indicate that TakaTuka is a very promising platform for WSN applications.

## References

[1] AREXX engineering. The NanoVM - Java for the AVR, http://www.harbaum.org
[2] Clausen, et al.: Java bytecode compression for low-end embedded systems. ACM Trans. Program. Lang. Syst. (2000)
[3] Gay, D., et al.: The nesc language: A holistic approach to networked embedded systems. In: ACM SIGPLAN PLDI (2003)
[4] Gregg, D., et al.: A fast java interpreter. In: The Workshop on Java (2001)
[5] Rayside, D., et al.: Compact java binaries for embedded systems. In: CASCON (1999)
[6] Saougkos, D., et al.: Revisiting java bytecode compression for embedded and mobile computing environments. IEEE Trans. Softw. Eng. (2007)
[7] Simon, D., et al.: Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In: ACM SIGPLAN VEE (2006)
[8] Gamma, E., et al.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)
[9] Chakeres, I., et al.: Dynamic MANET On-demand (DYMO) Routing. IETF (2008)
[10] Koshy, J., et al.: Vmstar: Sythesizing scalable runtime environments for sensor networks. In: SenSys (2005)
[11] Brouwers, N., et al.: Darjeeling, a feature-rich vm for the resource poor. In: SenSys (2009)
[12] Levis, P., et al.: TinyOS Programming. Cambridge University Press, Cambridge (2009)
[13] De Wang, S., et al.: Jato: A compact binary file format for java class. In: ICPADS (2001)
[14] Lindholm, T., et al.: The Java Virtual Machine Specification. Prentice-Hall, Englewood Cliffs (1999)
[15] Shi, Y., et al.: Virtual machine showdown: Stack versus registers. In: VEE (2005)
[16] Gnawali, O., Fonseca, R., Jamieson, K., Moss, D., Levis, P.: Collection tree protocol. In: Culler, D.E., Liu, J., Welsh, M. (eds.) SenSys, pp. 1–14. ACM, New York (2009)
[17] Sentilla, http://www.sentilla.com
[18] Crossbow Technology. Wireless Sensor Networks, http://www.xbow.com