Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks

Waqaas Munawar*‡, Muhammad Hamad Alizai†, Olaf Landsiedel†, Klaus Wehrle†
‡Computer Systems and Telematics, Freie Universität Berlin, Germany
†Distributed Systems Group, RWTH Aachen University, Germany
waqaas.munawar@fu-berlin.de,{hamad.alizai,olaf.landsiedel,klaus.wehrle}@rwth-aachen.de

***Abstract*—Long-term deployments of sensor networks in physically inaccessible environments make remote re-programmability of sensor nodes a necessity. Ranging from full image replacement to virtual machines, a variety of mechanisms exist today to deploy new software or to fix bugs in deployed systems. However, TinyOS - the current state of the art sensor node operating system - is still limited to full image replacement as nodes execute a statically-linked system-image generated at compilation time.**

In this paper we introduce Dynamic TinyOS to enable the dynamic exchange of software components and thus incrementally update the operating system and its applications. The core idea is to preserve the modularity of TinyOS, i.e. its componentization, which is lost during the normal compilation process, and enable runtime composition of TinyOS components on the sensor node. The proposed solution integrates seamlessly into the system architecture of TinyOS: It does not require any changes to the programming model of TinyOS and existing components can be reused transparently. Our evaluation shows that Dynamic TinyOS incurs a low performance overhead while keeping a smaller - up to one third - memory footprint than other comparable solutions.

I. Introduction

Wireless Sensor Networks (WSN) are envisioned to be deployed in the absence of permanent network infrastructure and in environments with limited or no human accessibility [1], [2], [3]. Hence, such deployments demand mechanisms to update nodes over the air: collecting nodes to apply updates is often tedious [1], [4] or even dangerous [2], [3]. Typical updates range from simple modifications, e.g., bug fixes, to the introduction of new functionality or re-tasking of sensor nodes.

Remote code-update schemes for sensor nodes meet two key challenges: (1) resource consumption and (2) integration into the system architecture. Energy, processing power, memory, and communication bandwidth are scarce resources on sensor nodes and their consumption needs to be limited. For example, full-image replacement [5], [6] results in high transmission costs while showing low processing demands. In contrast, modern approaches such as runtime linking of incremental updates [7], [8] or virtual machines [9], [10] reduce transmission costs while requiring post processing on the sensor node. Moreover, code-update schemes need to be transparently integrated into the system architecture to allow that existing applications can still be used and that users do not have to adapt to new programming models.

The contribution of this work is twofold: (1) we introduce extensions to TinyOS to create a dynamic operating system that supports efficient and dynamic adaptation of the OS and its applications at runtime and (2) we integrate these extensions transparently into the system architecture of TinyOS. We believe that systems like TinyOS, for which a large base of applications and communication protocols exists, require a transparent integration of code update schemes into the system architecture to ensure that no changes to this existing, large code base are required.

Our system design preserves the component model of TinyOS: TinyOS applications and the OS itself are built by connecting so called components. Components represent functional building blocks such as communication protocols, device drivers, or data analysis modules. During the default compilation process of TinyOS, these building blocks are converted into a single, static binary. While this enables code optimization and ensures a small memory footprint, it omits the modularity of the OS and its applications. In contrast, this work introduces dynamic extensions to TinyOS allowing the user to define TinyOS components that should be kept modular in the resulting executable. As a result, Dynamic TinyOS allows to replace these components dynamically at runtime.

The remainder of this paper is structured as follows. Section II discusses related work. We present a system overview in Section III and system design in Section IV. We evaluate Dynamic TinyOS in Section V and Section VI concludes the paper.

II. Related Work

Existing approaches to remote retasking of a sensor network can be classified into four main categories.

Full-Image Replacement: These techniques such as Xnp [5] or Deluge [6] operate by disseminating a new binary image of an application and the OS in the network. Since the image is compiled and linked afresh in every iteration, these solutions offer a very fine-grained control over the possible reconfigurations. However, these approaches result in bandwidth overhead as unchanged parts of an application need to be re-disseminated in the network.

*This work was conducted while the author was working with the Distributed Systems Group at RWTH Aachen University.

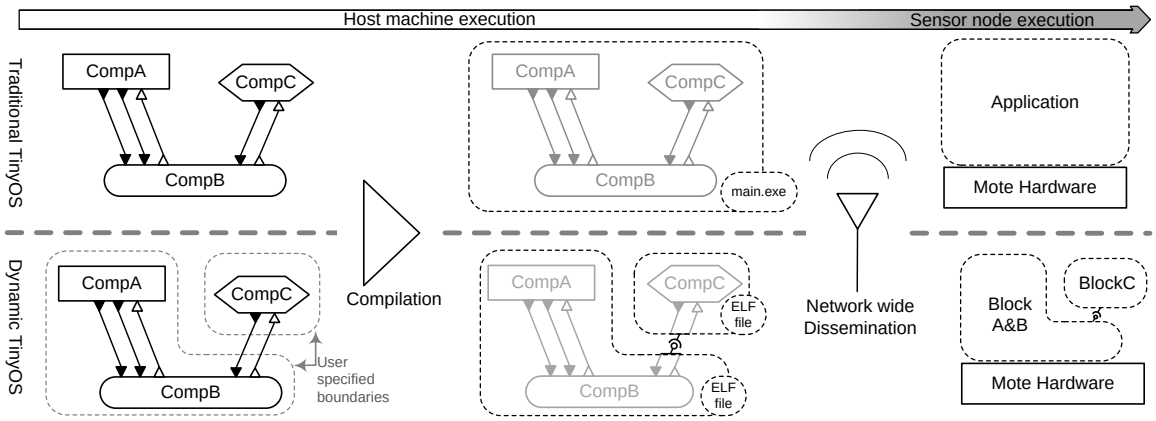

Fig. 1. Overview of code updates in Dynamic TinyOS in comparison with the traditional TinyOS: In the first step a user chooses boundaries among the building blocks of an application and the OS. In the second step, ELF files are generated according to the specified boundaries. These ELF file are disseminated in the network and linked on the mote to form the application.

Differential Image Replacement: Zephyr [7] and others [8], [11] disseminate the changes between an executable deployed in the network and a new image. While this reduces the bandwidth consumption, the fundamental drawback of image replacing remains: These approaches cannot utilize high-level knowledge of the application structure. Zephyr tries to minimize this effect by rerouting all function calls through an indirection table to mitigate the shifts in function locations between the old and the new version of an application. Although this increases the similarity between the two versions, it imposes additional memory and performance penalties.

Virtual Machines (VM): VMs such as Maté [9] and others [10], [12] reduce the energy-cost of disseminating new functionality in the network as VM code is commonly more compact than native code. However, virtual machines typically allow application updates only and interpret the VM code. Hence, they result in runtime overhead and decrease the lifetime of sensor nodes.

Dynamic Operating Systems: These, e.g., Contiki [13], SOS [14] and FiGaRo [15], provide the benefits of both image replacement and virtual machines i.e. fine grained code updates at low dissemination and run-time overhead. However, specific challenges remain: For example, SOS's design necessitates the use of position independent code, which, due to compiler limitations, is not fully supported on common WSN platforms. Contiki allows only one-way linking for loaded modules and hence obligates more energy-intensive, polling-based service routines for interrupts. Moreover, Contiki's architecture restricts possible reconfigurations to application components only.

Commonly, dynamic OSes follow a clean slate approach which causes hindrance in their wide scale adoption. Two notable exceptions are FlexCup [16] and TOSthreads [17], which are built on top of TinyOS. FlexCup offers dynamic adaptation for TinyOS based applications but lacks the support for new extensions to NesC, TinyOS programming language, and employs nonstandard tools. As a result, the non-standard toolsets need to be ported to a wide range of development platforms, making maintenance and the roll-out of new features time consuming. Similar to Contiki, the TOSthreads library and its linker limit code replacement to application components only. Moreover, it follows a polling based approach for kernel to application communication instead of NesC's well established and more efficient event based approach. Additionally, it introduces a new interface for users, rendering it difficult to adopt. Concluding, both these approaches are not transparent for end users, lack the support for reuse of TinyOS code, and cause a substantial increase in the steepness of the learning curve.

In contrast to existing work on dynamic OSs, this paper shows how an existing and well established OS can be transparently transformed into a dynamic OS without following a clean slate approach or introducing new programming models.

III. System Overview

In this section we briefly describe the overall architecture of Dynamic TinyOS. TinyOS based sensor network applications consists of a large selection of individual software components which are 'wired' together to achieve the desired functionality. In the standard TinyOS compilation process, these building blocks are mashed-up to form a single, monolithic binary-image of the application. However, as each component provides a dedicated functionality to the overall system, updates such as the deployment of a new functionality or a bug fix are commonly limited to a small number of neighboring components or even a single component. Hence, the modularity of TinyOS forms a natural starting point for a dynamic operating system: Dynamic TinyOS alters the compilation process of TinyOS to preserve user selected parts of the component structure across the compilation phase. The result is an executable consisting of multiple, replaceable objects. Hence, during deployment, updates of applications or of the OS itself can be disseminated in the network to replace existing objects on the sensor node.

Code updates in Dynamic TinyOS work in three phases, see Figure 1: (1) Via extensions to the NesC compiler of

TinyOS we compile components, i.e., applications and system components, into multiple objects. As a result, the component based structure of the TinyOS application is preserved during the compilation process. (2) Using a standard dissemination algorithm[1], such as the one of Deluge [6] or others [18], [19], updates - i.e. binary objects - are transferred to the sensor node over the radio. (3) A thin runtime on the node stores these updates and integrates new components into applications or the OS.

Moreover, Dynamic TinyOS allows users to define the granularity of replacements. Hence, a user can combine multiple TinyOS components into a single object. While this increases the size of updates, it reduces the overhead of run-time linking on the sensor node and enables compiler optimizations inside this object. As an example, Figure 1 shows the process of dividing an application. It can be one of the applications shown in Table I with its respective components. Here the application is divided into two blocks, one containing components A and B and the other containing component C. Any updates to component C will only require the retransmission of this single component for updates, while updates to A require the block containing A and B to be disseminated in the network. Thus, the tradeoff between transmission energy and linking overhead can be adapted based on expected future application and deployment requirements.

IV. Dynamic TinyOS

In this section we discuss the architecture of Dynamic TinyOS in detail. It consists of two main components: (1) On the host, we isolate a single TinyOS component or a group of components and compile them into an ELF object. (2) We provide *Tiny Manager*, a runtime system executing on the sensor node. It handles storage and integration of new components, i.e., code updates. These new ELF objects are linked into an executable binary-image and loaded in program memory. Next to the discussion of these two core components of Dynamic TinyOS, we conclude this section by presenting optimizations for ELF objects to minimize transmission and linking overhead.

A. Host-Side System

First, Dynamic TinyOS compiles an application and the OS core into separate ELF files based on user specified boundaries for incremental updates. Compiling parts of a TinyOS application in isolation from the rest has two side effects: (1) It introduces ambiguities, such as the parametrization of NesC generics and default event handlers, and (2) it limits compiler optimizations. To address the first issue, Dynamic TinyOS provides a compiler extension, so called component isolation, which resolves these ambiguities and enables automated compilation of TinyOS components into solitary ELF objects. Dynamic TinyOS addresses the second issue by allowing the user to group multiple TinyOS components into a single object limiting modularization to user required parts. While these larger objects increase optimization possibilities, such as code inlining and loop unrollments, they increase the size of updates. Hence, Dynamic TinyOS allows users to balance the cost of updates and their performance penalties.

[1]Our approach is independent of any dissemination algorithm and well established code dissemination algorithms are available. A thorough exploration of these algorithms is beyond the scope of discussion in this paper.

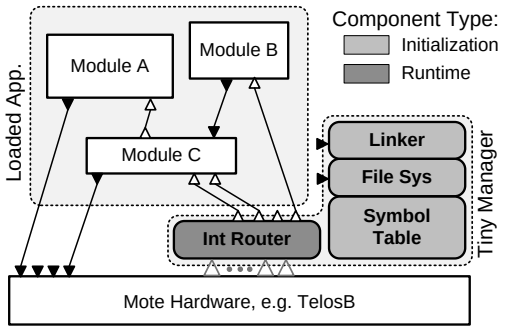

Fig. 2. Architectural elements of *Tiny Manager*. Only the interrupt router is active during the normal execution of a loaded application. Linker, file system and symbol table handle storage and integration of newly received code.

1) Component Isolation: The main issues faced during isolation of TinyOS components is the non-availability of system information which is hidden in parts of the application that are not being compiled at the moment. For example, the timer dispatch component needs to be parameterized with the number of timers used in the OS and applications. Similarly, the scheduler needs to be parameterized with the number of threads in the system. *Component Isolation* of Dynamic TinyOS parameterizes such components by collecting information from other modules in the system and user requirements with which it configures the NesC compiler of TinyOS. This parameterization consists of two main parts for each component to be isolated: a component-wrapper and an application side place holder. The component wrapper ensures that the component being isolated is provided with the required knowledge of the rest of the application for correct compilation. Likewise, the application side place-holder ensures that the application gets the required knowledge about the component which will be linked-in at runtime. During this process, the actual source code of both the application and its component is not changed. This transparent integration allows the reuse of existing TinyOS based applications and seamless integration of Dynamic TinyOS into the existing TinyOS skeleton, thereby remaining transparent to the application developer.

2) Component Over-Provisioning: Component over-provisioning allows to provide additional functionality for expected future updates. For example, the OS core can provide additional timers or slots for additional threads expected to be required by future deployments of new functionality. Additionally, over-provisioning can be used to configure a *base block* to provide the functionality needed by typical sensor network applications, i.e., consisting of timers, scheduler, radio and other hardware drivers. This design ensures that a currently deployed application can be changed to radically different tasks by merely communicating the user implemented part of the new application (see our evaluation

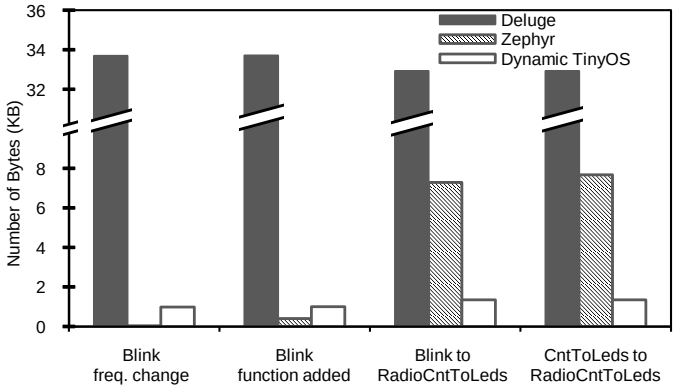

Fig. 3. Performance in common software update scenarios. Zephyr results in smaller updates for very small changes in application. In contrast, Dynamic TinyOS has a stable update-size because it operates at the component level. It outperforms Zephyr in the case of bigger changes in applications such as the addition of a new component.

in Section V for an example).

B. Node Runtime: Tiny Manager

Once updates have been received on the sensor nodes, the dissemination protocol invokes the linker to integrate the received modules and place the resulting new binary image in the program memory of the sensor node. To accomplish this, Dynamic TinyOS provides a runtime on the sensor node, called Tiny Manager (see Figure 2). It consists of the following four main components:

- **File System:** The file system provides storage capabilities for received ELF objects.
- **Linker:** The linker is responsible for linking newly received ELF objects with the program base and placing the resulting binary in code memory.
- **Global Symbol Table:** A global symbol table maintains the symbols offered by each module and resolves dependencies among individual modules.
- **Interrupt Router:** Unlike a compile-time linker, a run-time linker does not have flexibility in the placement of code segments. Hence, location dependent code such as interrupt handlers are replaced by a place holder, called interrupt router, which forwards interrupts to their corresponding handlers placed dynamically at run-time.

Apart from the interrupt router, all components are inactive during the normal execution of application. This minimizes the runtime overhead of the *Tiny Manager*.

C. ELF Optimizations

After discussing the design of Dynamic TinyOS in detail, we discuss optimizations to ELF files to reduce their size. These optimizations reduce transmission overhead and storage requirements which are both scarce resources on sensor nodes.

The ELF format, though a widely used standard, is not optimized for low power processors. For example, the string table, which holds the names of all symbols in a ELF file, can account for a large fraction of the file. Due to the naming

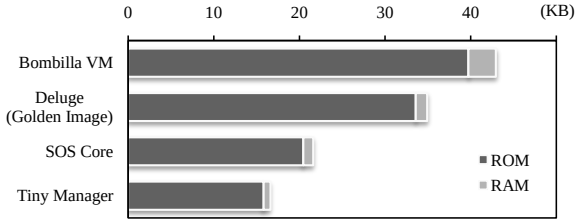

Fig. 4. Memory-footprint comparison for *Dynamic TinyOS*. Tiny Manager only utilizes 7.7% of the RAM and 32% of the internal flash ROM on TelosB platform, which is significantly less than all other comparable solutions

schemes of the NCC compiler in TinyOS, function names often tend to be quite long, on average about 80 characters each. We decrease the size of the symbol names down to 3 characters by replacing each symbol name with a unique string based on an alphanumeric counter. The mapping of these names is stored in a database on the host and is used when recompiling and updating parts of the application. This procedure results in: (1) significant reduction in the size of ELF file, (2) reduction in the size of the symbol table and (3) reduction in number of string comparison operations during linking on the sensor node.

As second optimization we split the symbol table on the sensor node into two sub-tables: one contains static core symbols and the other is filled at runtime with the symbols of dynamically loaded ELF modules. The static part is created at compilation time and placed sorted in ROM. As a result, this table allows a fast binary search among the symbols, increasing the energy efficiency of the linking process.

These two optimizations enhance the processing speed, resulting in energy savings of up to 66% when compared to the original ELF file while not changing the structure of ELF files themselves. Hence, our optimizations do not require customized tools, allowing the use of standard tool chains, and result in a low maintenance and porting effort.

V. Experimental Evaluation

After discussing the design of Dynamic TinyOS we next compare its performance with existing approaches for code updates in TinyOS, such as Deluge and, where possible[2], Zephyr and Maté. The evaluation focuses on key factors such as the size of updates, energy consumption, memory footprint, and processing overhead. We implemented Dynamic TinyOS for the TelosB platform [20] and the recent 2.1 release of TinyOS [21]. Hence, our comparative evaluation covers a broad range of standard applications from the TinyOS repository, underlining the feasibility of our approach.

A. Size of Updates

First, we evaluate the size of updates in Dynamic TinyOS to determine its energy and storage requirements. Figure 3 shows the results for our approach in comparison to Deluge

[2]Zephyr is not (yet) open-source. Hence, – when available – our comparison relies on the same benchmarks as used by Zephyr to establish a base for a fair comparison.

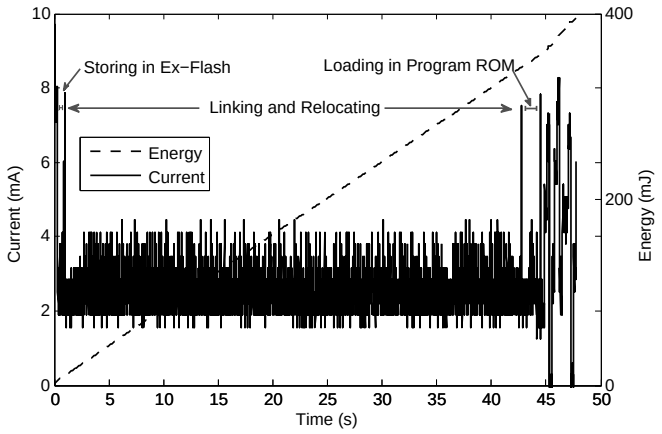

Fig. 5. Current draw and energy utilization during processing and loading of the BlinkTask application on the TelosB platform. The peaks, generated by turning on all onboard LEDs simultaneously, mark the boundaries between the different operations.

Application	Component	Size (B)	Tx. Energy(mJ)	Deluge Size	Saving Factor
Blink	OS-Comp	6616	465.1	33726	5.1
	Blink	824	57.93		40.9
	Leds	1728	121.48		19.5
	Timer	5424	381.31		6.2
	Scheduler	1980	139.19		17
BlinkTask	OS-Comp	6640	466.79	33726	5
	Blink	992	69.74		34
	Leds	1728	121.48		19.5
	Timer	5424	381.31		6.2
	Scheduler	2356	165.63		14.3
Radio-CntToLeds	OS-Comp	28232	1984.71	33954	1.2
	Radio	1352	95.05		25.1
	Leds	1728	121.48		19.6
	Timer	4772	335.47		7.1
	Scheduler	3092	217.37		10.9
Sense	OS-Comp	17040	1197.91	34074	2
	Sense	940	66.08		36.25
	Leds	1728	121.48		19.7
	Timer	6296	442.61		5.4
	Scheduler	2576	181.09		13.2
Oscillo-scope	OS-Comp	39328	2764.75	34504	0.87
	Oscilloscope	2008	141.16		17.1
	Leds	1720	120.91		20.0
	Scheduler	3728	262.07		9.25

TABLE I
SAVINGS IN TRANSFER ENERGY DUE TO INCREMENTAL UPDATES

and Zephyr for different software update scenarios ranging from a simple timer-frequency change in the Blink application to the retasking of sensor nodes with a completely new application. The results show that Zephyr, as it is based on byte level comparison, performs better for very small changes in an application, such as the addition of a small function to a component. However, Dynamic TinyOS shows consistent update-sizes and outperforms existing approaches for component level changes, such as addition of a new component to the existing application. This is because it preserves the high level structural knowledge of an application and its components. Hence, updating from the CntToLeds to the RadioCntToLeds application only requires communicating the main application component, while radio, timer and led components can be reused.

B. Memory Footprint

Due to a thin runtime component, i.e., Tiny Manager, the memory footprint of Dynamic TinyOS is small in comparison to popular existing solutions (see Figure 4). On the TelosB platform, it consumes only 7.7% of RAM and 32% of the program memory. Furthermore, the external flash memory is completely available for the file system and 'Golden Images'. Hence, it leaves the majority of the storage resources for the OS, applications and code updates.

C. Energy Consumption

The two main factors that contribute to the overall energy overhead of Dynamic TinyOS are: (1) dissemination of component updates and (2) linking of newly received components with the application on the sensor node.

1) Code Dissemination: We evaluate the energy consumption during the dissemination process of Dynamic TinyOS and compare our results with Deluge for a set of representative applications from the TinyOS repository (see Table I). These applications utilize a broad range of TinyOS system components and protocols - MAC, timers, LEDs, radio, and sensing hardware - required to drive the sensor hardware-platform, allowing us to comprehensively validate our results. It is fair to conclude that Dynamic TinyOS consumes significantly less transmission energy - up to a factor of 40 - than Deluge. However, reducing the size of our updates introduces processing overhead at the sensor node as discussed in the next section.

2) Processing: Processing an update consists of three steps: (1) storage in the external flash, i.e., file system, (2) linking and relocating, and (3) loading into program memory (see Figure 5). The energy consumption of this processing and loading of an updated component does not solely depend on the size of the component but also on the symbol dependencies and the number of relocations that need to be performed.

Deluge has a constant energy overhead because it always disseminates the complete application and OS image. In contrast, the energy overhead of Dynamic TinyOS depends on the size of the components to be updated and the required processing on the sensor node. Overall, Figure 6 shows that Dynamic TinyOS outperforms Deluge in terms of the overall energy required for code dissemination and processing of updates.

D. Runtime Performance Overhead

The changes required to enable incremental code updates in TinyOS effect two main areas: (1) the compilation process and (2) the inclusion of an additional runtime layer, i.e., TinyManager.

1) Compiler Optimizations: Compiling parts of an application in isolation reduces the overall code optimization possibilities for a compiler. Moreover, setting explicit boundaries between application components can result in additional calls to those functions which otherwise might have been *inline*d. To

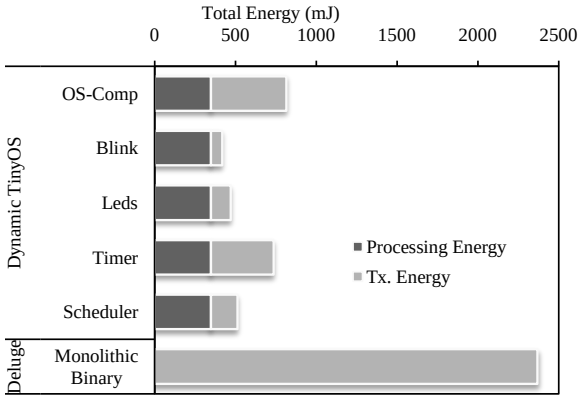

Fig. 6. Energy overhead comparison of *Dynamic TinyOS* with Deluge. Dynamic TinyOS results in significantly less overhead than Deluge for all the update scenarios. Deluge has a constant overhead for all update scenarios.

stress-test the impact of isolated compilation of components, we use two benchmarks: (a) A syntactic benchmark, which uses the TestScheduler application from the TinyOS repository, as a sanity check for our approach. To evaluate our approach from the worst-case point of view, we modified this application to make 10,000 cross component calls to the Scheduler. (b) An application that calculates a 256 point FFT.

Table II shows that Dynamic TinyOS consumes slightly more code memory when compared with the traditional TinyOS. Furthermore, the runtime performance overhead of Dynamic TinyOS is very small: While showing a less than 10% worst case overhead, it takes the same execution time for a computationally intensive real-world algorithm, i.e., FFT.

2) Interrupt Re-Routing Overhead: During normal application execution, i.e., when no are updates processed, only the interrupt routing component of *Tiny Manager* is active. It introduces a short delay in the processing of interrupts. On the TelosB platform, the worst case delay is 23 instruction cycles – equivalent to processing required for copying eight bytes in memory. No performance depreciation is caused by other components and hence code execution in Dynamic TinyOS remains native.

Overall, our evaluation shows that Dynamic TinyOS outperforms other approaches for code updates in TinyOS and even shows a very small overhead when compared to a static TinyOS binary.

VI. Conclusions

In this paper, we presented Dynamic TinyOS to enable fine grained code updates of deployed TinyOS based applications. It outperforms other approaches for code updates in terms of memory footprint and energy consumption while resulting in a minimal run-time overhead. Even, when compared to a traditional, static TinyOS binary, it shows only a very small memory and performance overhead. Furthermore, Dynamic TinyOS provides an unprecedented level of flexibility in the granularity of the possible reconfigurations of both applications and the OS core at runtime.

Application	Dynamic TinyOS		Traditional TinyOS	
	ROM (B)	Time	ROM (B)	Time
TestScheduler	1810	504ms	1754	460ms
FFT	14718	5.320s	14232	5.320s

TABLE II
Impact of compiler optimizations on the memory size and execution time

Overall, Dynamic TinyOS adds a minimal performance overhead and integrates transparently into the existing TinyOS system architecture to enable code-reuse and to remain transparent to end users. We believe that the ability to load modules dynamically – as introduced by Dynamic TinyOS – is a very logical evolution of TinyOS in it's current state.

References

[1] D. Pompili, T. Melodia, and I. F. Akyildiz, "Deployment analysis in underwater acoustic wireless sensor networks," in *WUWNet '06*, 2006.

[2] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein, "Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet," in *ASPLOS-X*, 2002.

[3] G. Werner-Allen, K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees, "Deploying a wireless sensor network on an active volcano," *IEEE Internet Computing*, vol. 10, no. 2, 2006.

[4] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin, "Habitat monitoring with sensor networks," *Commun. ACM*, vol. 47, no. 6, 2004.

[5] J. Jeong, S. Kim, and A. Broad, "Network reprogramming," Aug 12, 2003. [Online]. Available: http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf

[6] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *SenSys '04*, 2004.

[7] R. K. Panta, S. Bagchi, and S. Midkiff, "Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation," in *USENIX '09*, 2009.

[8] J. Jeong and D. Culler, "Incremental network programming for wireless sensors," in *IEEE Secon '04*, Santa Clara, USA, 2004.

[9] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," in *ASPLOS-X*. ACM, 2002.

[10] R. Müller, G. Alonso, and D. Kossmann, "A virtual machine for sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 145–158, 2007.

[11] N. Reijers and K. Langendoen, "Efficient code distribution in wireless sensor networks," in *WSNA '03*, 2003.

[12] N. Brouwers, P. Corke, and K. Langendoen, "A java compatible virtual machine for wireless sensor nodes (demo abstract)," in *SenSys '08*, 2008.

[13] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *LCN '04*, 2004.

[14] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *MobiSys '05*, 2005.

[15] L. Mottola, G. P. Picco, and A. A. Sheikh, "Figaro: Fine-grained software reconfiguration for wireless sensor networks," in *EWSN*, 2008.

[16] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "Flexcup: A flexible and efficient code update mechanism for sensor networks," in *EWSN '06*, 2006.

[17] K. Klues, C.-J. M. Liang, J. yeup Paek, R. Musaloiu-E., P. Levis, A. Terzis, and R. Govindan, "TOSThreads: Safe and Non-Invasive Preemption in TinyOS," in *Sensys '09*, 2009.

[18] T. Stathopoulos, J. Heidemann, and D. Estrin, "A remote code update mechanism for wireless sensor networks," *University of California, LA, Tech. Rep. CENS-TR-30*, 2003.

[19] P. Levis, N. Patel, S. Shenker, and D. Culler, "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *NSDI '04*, 2004.

[20] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *IPSN '05*, Los Angeles, California, 2005.

[21] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "The emergence of networking abstractions and techniques in tinyos," in *NSDI'04*, 2004.