Live Code Update for IoT Devices in Energy Harvesting Environments

Chi Zhang, Wonsun Ahn, Youtao Zhang, and Bruce R. Childers
Department of Computer Science, University of Pittsburgh
Pittsburgh, Pennsylvania 15260 USA
Email: {chizhang, wahn, zhangyt, childers}@cs.pitt.edu

***Abstract*—The number of Internet of Things (IoT) devices is exhibiting explosive growth. These devices are often closely coupled to the physical world, and may harvest energy as a power source, which imposes particularly stringent operating constraints. Like any programmable system, IoT devices may need software updates to fix bugs, add functionality, or improve computational capability. This paper proposes novel strategies to update deployed code for IoT energy-harvesting devices based on in-place code updating and code trampolines, which effectively eliminate system down time and minimize resource demands for updates. We show that our schemes, on average, reduce the number of nonvolatile memory writes by 99% and code transmission cost by 78%.**

I. Introduction

The Internet of Things (IoT) is a novel computing paradigm that couples sensing devices, computing nodes, communication devices with various types of objects in physical world for data collection, exchange, and remote control. IoT devices often have very tight constraints on cost, form factor, and power/energy consumption. For example, it is not realistic to attach a power cord to devices implanted inside the human body, and exchanging batteries for such devices sometimes can be life-threatening. These devices often rely on ambient power sources such as wireless energy, RF energy, solar energy, and piezoelectric energy.

The ambient power is not only sparse but also often unreliable, which makes it necessary to equip these devices with non-volatile memory to store program state in order to ensure forward progress. Often only a couple of instructions can be executed per power cycle after which state has to be stored. To enable frequent and fine-grained storage of program state, FeRAM or ReRAM are preferred over Phase change memory (PCM) or NAND flash as the latter have large write power and NAND flash uses page-level write granularity [1].

Another critical concern is how to deliver code updates to the energy-harvesting devices post-deployment. Traditional post-deployment code update schemes focus mainly on reducing the amount of data transferred over the wireless network. This includes proposals involving incremental updates, modular designs, and network encoding. For example, incremental update schemes attempt to minimize the code transferred to a device by sending only the "delta" difference between the old and new images, instead of sending the entire image. The new image then is constructed from the old image and the delta [2], [3], [4], [5], [6].

Regardless of the code update scheme, past approaches have the following steps: 1) transmit the code update, 2) (optional) construct a new image if the code update was a delta, and 3) reboot the device using the new code image. We call this approach *image rewrite*. This approach has several drawbacks.

First, it degrades the QoS level of the device. Image rewriting requires rebooting a device and relaunching the application during which the device is unresponsive to external events. This delay is down time during which the device is unusable. The down time can last hours to even days in energy harvesting processors, depending on the available energy at the time of the reboot. For example, in a recent deployment of sensors on the Reventador Volcano, reboots from software misconfigurations and bugs led to a 3-day network outage, reducing mean node uptime from >90% to 69% [7], [8].

Second, it takes longer for code updates to be delivered to devices because existing image rewrite approaches do not work well with incremental updating. Code updates involving addition or deletion of code cause "shifts" in the addresses of functions and global variables in the new image compared to the old image. This shift can result in numerous differences in the targets of function call instructions and global variable accesses between the old and new images, even for seemingly trivial updates. This can greatly increase the delta that needs to be transmitted across the network.

Third, image rewriting takes longer for code updates to be applied once the patches are delivered because the whole new code image must be rewritten to code memory, even for small updates. For processors that rely on harvested energy, this rewrite process may take a significant time to complete, again depending on the energy available. Many iterative code update scenarios, such as debugging and software tuning, depend on a short turn-around time between code modification and application. For these scenarios, the increase in update delivery time and application time reduces the usability of the devices.

Lastly, image rewriting increases the cost of the device by requiring larger code memory. Image rewrite approaches require the old image to be running while constructing or downloading the new image. Thus, code memory capacity must be at least twice the size of the code image. This not only increases device cost but also impacts the power consumption. A larger code memory translates to more standby leakage power and access power due to driving longer bitlines.

In this paper, we address live code updates for IoT devices in energy harvesting environments. We avoid the above problems by updating the code image *in-place* through patches, while

the code is live and still executing.

In-place updating solves or reduces the problems with image rewriting. It does not suffer from QoS degradation since no reboot is required — all patches are performed on live code. It also reduces code update delivery cost for incremental approaches since there is no increase in the delta size due to code shifting — all patches are performed without moving existing code. At the same time, it drastically reduces the cost to apply the code update to code memory since only new code in the form of patches have to be written to code memory, and not the entire image. Lastly, it does not require code memory to hold two images at the same time.

However, in-place updating introduces a new code consistency problem that does not exist with image rewrite updating. The problem is that the image being updated is live code and during the course of an update, the code can fall into an inconsistent state during which it should not be allowed to execute. Hence, all event handling that may execute the code under update must be temporarily halted until the update completes. If not done intelligently, in-place code updating can reintroduce device down time.

Also, in-place updating may require modifications to be done to the code update software itself which is already running. For this update scenario, the modification somehow has to happen *atomically*, even when it involves multiple patches to code memory. The program code cannot be left in an inconsistent state at any point during execution. We solve this problem using a *code trampoline* mechanism.

We make the following contributions in this article:

- We propose a novel in-place patching strategy for the three code update scenarios of insertion, deletion, and modification that minimizes the down time of a device.
- We propose an improvement over the in-place patching strategy using trampolines to permit updates to take effect atomically with a single store to memory.
- We evaluate these two strategies on device down time, code update delivery cost, code memory update cost, and runtime performance overhead.

II. Background

Code updates on IoT devices are performed by first generating a new image at the base station and transferring the image over a wireless network to target devices. While some early techniques simply transfer the entire image [9], [10], recent approaches use incremental code updates to transfer only the difference between the old image in the device and the new image [2], [3], [4], [5]. This difference is expressed as a "delta script" that contains commands to copy a range of addresses from the old to the new image, or commands to insert new code sent with the script. Since new code is a (typically) small subset of the new image, sending delta scripts is usually much cheaper than sending the entire image.

A major issue with incremental code updates are code shifts caused by insertion or deletion of code. Shifts in code can change the positions of functions and global variables between the old and new images. Thus, calls to these functions or accesses to the global variables in the new image have different target addresses and must be sent as new code in the delta script. Previous work has proposed several methods to prevent code shifts or modifications to target addresses even with shifts. This includes: replacing function calls to jumps to function call indirection tables to prevent call target modification [2], pinning the addresses of global variables [3], and sending a relocation table with the delta script [4], [5].

Our in-place update scheme is a form of incremental code update in that only a delta script containing the patch is sent to a target device (rather than the whole image). However, unlike other incremental update proposals, the delta is applied in-place on live code instead of being used to construct a new image. Also, since the delta is applied in-place, there is no shifting involved. Hence, the problem of shifting is inherently solved without reverting to suboptimal solutions from past proposals to prevent address shifting.

III. Design and Implementation

Unlike image rewriting, in-place code needs to handle the situation that code being updated could be live and actively executing. A code update typically involves multiple patches during which execution of code being patched must be suspended to avoid inconsistent code. Halting the device for this period of time can cause QoS problems, similar to rebooting causes for image rewriting.

Hence, the goal of in-place code updating is to minimize the set of code memory writes during which the code falls into an inconsistent state. We call this set of writes the *atomic update set*, since the writes must happen atomically to execution of the code. If done carefully, the atomic update set can be kept to a small subset of the entire set of writes that need to happen.

In this section, we present two code patch strategies to minimize the atomic update set. The first strategy, *in-place patching*, can reduce the atomic update set to be proportional only to the number of patch locations instead of the size of the patch. The second strategy, *in-place patching with trampolines*, can reduce the atomic update set to just one write.

A. In-Place Patching

Figure 1 shows in-place patching for insertion, deletion and modification code updates. The white boxes represent the original code and the dark gray boxes represent code that is part of the atomic update set. The light gray boxes represent code that is part of the update set but does not have to be updated atomically. The dotted boxes represent the code that is being inserted, deleted, or modified.

Code insertion is performed by inserting a jump from the location of the insertion point in the original image to the inserted code. At the end of the inserted code, a jump back to the instruction immediately following the insertion point is added to continue execution in the original image. Note that the jump instruction will overwrite the instruction that is already at the insertion point; the original instruction is moved to the beginning of the inserted code. This is analogous to how jumps to breakpoint handlers are implemented with fast breakpoints [11].

Code deletion is performed by inserting a jump from the location of the first deleted instruction in the original image to another jump instruction. The second jump jumps back to the original image to the instruction immediately following the last deleted instruction. Code modification is performed in a similar fashion except, after execution of the modified code, control flow jumps to the point after the old code in the original image. In practice, a code update involving multiple patches has two phases:

1) **Phase 1**: All new or modified code in light gray boxes is written to free space in code memory (i.e., unused by the current code image). The code is never inconsistent during this phase since no writes are performed on the original image; hence, the light gray color.
2) **Phase 2**: All jumps to the code written in Phase 1 are written atomically. Each of these jumps modify the functionality of the old image and have the potential to put the code into an inconsistent state. The execution of this code must be prevented until all writes complete; hence, the dark gray color.

The atomic update set of jumps to patched code is proportional to the number of *locations* that need to be patched in the original image, rather the size of patches themselves. Specifically, the atomic update set does not contain the actual instructions that are inserted or modified as part of the patch. Also, while code deletion could be done by overwriting the deleted instructions with no-ops, a jump is inserted instead to minimize the atomic update set. Otherwise, the no-ops would be part of the atomic update set.

Fig. 1: In-place patching for different code update scenarios: (a) Insertion, (b) Deletion, and (c) Modification of code.

B. In-Place Patching with Trampolines

While in-place patching reduces the atomic update set, it can still involve dozens of writes, depending on the complexity of the update. During these writes, the device is fully or partially disabled. Moreover, if the code update involves functionality used by the code update software itself (which is executing to do the update), inconsistent code may be exposed. Hence, the code update software could be forced to execute inconsistent code, leading to correctness problems.

Our second proposal for in-place code patching uses a jump trampoline to reduce the atomic update set to just one write. To this end, the patch strategy operates in three phases: trampoline insertion, trampoline target update, and trampoline removal. In the following sections, we use an example where a code insertion, a code deletion, and code modification are performed as part of the same code update.

1) Phase 1: Trampoline Insertion: Figure 2 shows the original code patched by an insertion, a deletion, and a modification after Phase 1 is completed. The code in dark gray is the atomic update set, while the code in light gray are the remaining updates that do not need to be atomic.

Note the similarities with the first in-place patching approach in Figure 1. However, jumps in the original image are to code snippets, called *trampolines*, rather than patched code. Each trampoline has an identical structure; each has three instructions: 1) the load of a `base` address to a jump target table, 2) the load of a target address from the table an offset from the base address, and 3) a jump to the target address. There are two jump target tables: 1) `orig_targets` that contain jump targets back to the original code immediately following the jump to the trampoline, and 2) `patch_targets` that contain jump targets to the patched code.

`Base` initially has `orig_targets`, meaning all trampolines use the `orig_targets` table for their jump targets. Since these are jumps back to the original code, the trampolines at this phase do not change the functionality of the code. Therefore, the trampolines can be inserted in any order and interleaved with code execution without worrying about consistency, as long as they are inserted after setting up the jump target tables. Unlike Figure 1, the jumps to the trampolines are not part of the atomic update set.

Fig. 2: Phase 1: Trampoline Insertion.

2) Phase 2: Trampoline Target Update: Figure 3 shows the code after Phase 2 is completed. The only update at this phase is the change of `base` from `orig_targets` to `patch_targets`, in dark gray. This single change updates the jump targets of all trampolines to the patch code in one shot, and no further updates are required to apply the patch. Thus, this update is the only one in the atomic update set across the entire patch process.

3) Phase 3: Trampoline Removal: While patching could stop at Phase 2, the resulting code requires additional execution of the trampoline instructions to achieve the same functionality as the basic approach (compare to Figure 1). Phase 3 streamlines the code further.

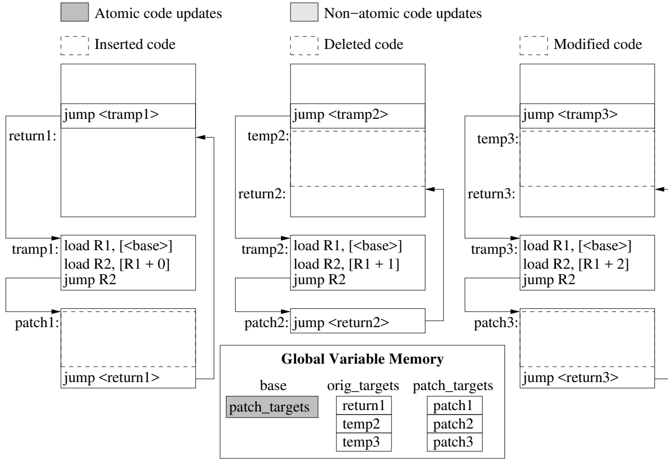

Fig. 3: Phase 2: Trampoline Target Update.

Figure 4 shows the code after Phase 3 is completed. In this phase, all jump targets in the original code are updated so that they jump directly to the patch code rather than take a detour through the trampoline code. Note that each of these updates does not change the functionality of the code, and they are not part of the atomic update set. When all is done, the code looks exactly the same as in Figure 1.

When the trampolines are no longer required, these same trampolines can be reused for the next patch. Note that the trampolines are identical across patches and can be reused with no modification.

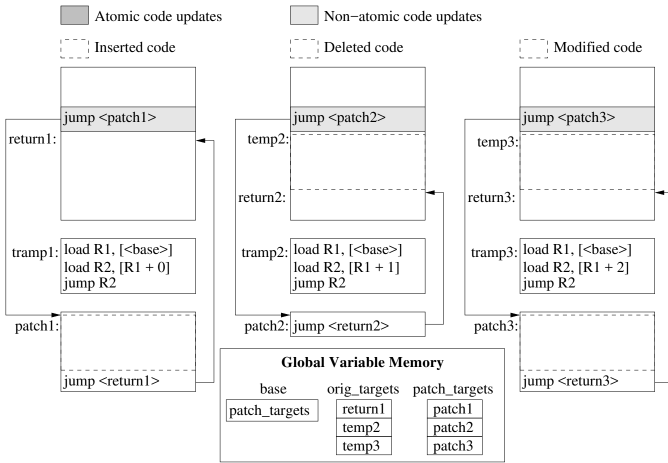

Fig. 4: Phase 3: Trampoline Removal.

IV. Discussion

Some code updates may involve addition or deletion of data variables, which can cause changes to data memory layout. Further, some code updates may change the semantic meaning of variables such that the old values are no longer usable. In these cases, our techniques can support the changes by rebooting or restarting the device and/or application. Fortunately, many code updates do not involve changes to data and the reboot can be avoided. Even if changes are made to data, they are often to local variables in functions that are not part of the current call stack. Since these variables are not currently being used, they can be safely modified without a reboot. Only when the change involves a local variable in the current call stack or a global variable is a reboot required. Since the current call stack typically contains only code related to the code update software, this is a rare event.

Fragmentation in code memory due to patching is also an issue. When deleting or modifying code, patching can leave "holes" in the original image (see Figure 1(b) and (c)). While these holes can be reused to store future patches, it is often hard to find a perfect fit. Eventually, there may come a time when patches can no longer be performed due to lack of code memory. At this point, these holes need to be compressed, which could possibly be done with code patches. However, we leave the subject of code defragmentation for future work.

V. Evaluation

A. Methodology

We evaluate in-place patching on several TinyOS benchmarks shown in Table I. These benchmarks exercise various components of a sensor, such as periodic sensing and broadcasts. We apply code patches to the benchmarks for scenarios in Table II. We use TinyOS 2.1.2. On these nine scenarios, we test four approaches, including two previous techniques:

- *RSync*: This is an image rewrite incremental code update strategy based on RSync delta patching [12]. It uses a delta script composed of two types of commands to construct the new image from the old image: COPY <old addr> <new addr> <len> and INSERT <addr> <len> <new code>. The former command copies a length of data from the old image address to the new image address. It is used to copy unmodified code. The latter command inserts provided code into the new image address. It is used to insert new code.
- *Zephyr*: This incremental image rewriting strategy improves on *RSync*. It use function indirection tables to reduce commands in the delta script from code shifting [2].
- *In-place*: This is the in-place patching given in Section III-A. The delta script uses three commands for the code updates in Figure 1: INSERT <addr> <len> <new code>, DELETE <addr> <len>, and MODIFY <addr> <len> <new code>. The code update software performs patching for each command at the given addresses.
- *Trampolines*: This is our second proposed code update strategy that improves upon in-place patching using trampolines; see Section III-B.

To determine delta scripts, we first generate the assembly code for the original benchmark and the patched benchmark for each test case using the NesC compiler. For *In-place* and *Trampolines*, we manually insert patches and trampolines at the assembly level. We use a custom analysis program to extract the delta script for each approach by comparing the two assembly codes. Using these scripts, we evaluate the each device down time, total code update time, and runtime performance of each approach.

To evaluate device down time and code update time, we measure how much energy is spent in the duration, since in

Benchmark	Source	Description
Blink	TinyOS	It starts a 1Hz timer and toggles the red LED every time it fires.
MultihopOscilloscope	TinyOS	A simple data-collection demo. It periodically samples the default sensor and broadcast a message every reading.
Oscilloscope	TinyOS	A simple data-collection demo. It periodically samples the default sensor and broadcast a message every 10 readings.
RadioCountToLeds	TinyOS	It maintains a 4Hz counter, broadcasting its value in an AM packet every time it gets updated.
RadioSenseToLeds	TinyOS	It samples a platform's default sensor at 4Hz and broadcasts this value in an AM packet.

TABLE I: Benchmark applications.

Cases	Update Details	No. of Patches
#1	In Blink: Change timer0 frequency from 250 to 50	1
#2	In Oscilloscope : insert one local variable and one use,*CancelTask_runTask	17
#3	In MultihopOscilloscope : insert one local var and use it within a loop in function MeasureClockC_Init	17
#4	In RadioCountToLeds : insert one local var into func,*nextPacket and use it twice - within and outside a loop	11
#5	In RadioSenseToLeds : Change one instruction (+ changed to -)	1
#6	In Blink: remove one function call statement in NesC code level	2
#7	In Blink: remove entire timer 0	33
#8	In RadioCountToLeds : add an else branch	5
#9	In Oscilloscope : insert a global var, and use it in three different functions	38

TABLE II: Code update test cases.

energy harvesting devices, the time required for performing a task is dictated by the time spent collecting energy for that task. Specifically, we first characterize the size of the atomic write set, the size of the delta script, and the number of bytes written to code memory. Then, based on these metrics, we calculate the energy expended when using a typical energy harvesting IoT device comprised of a MSP430FR5739 [13] microcontroller coupled with a AX5243 [14] wireless transceiver. The FeRAM used in MSP430FR5739 for code memory allows the microcontroller to perform code updates with very little write energy (4.25 nJ / byte). Also, unlike NAND flash, it can perform word granularity writes which enables efficient application of fine-grained in-place code patches. The runtime performance cost is estimated by running the benchmark images on the Avrora simulator [15] and using the instruction profiler to count the number of instructions with the different approaches.

B. Device Down Time

One of the most important advantages of in-place update is that it reduces or eliminates device down time. Image rewriting approaches suffer prolonged device down time while the OS reboots from the new image. In comparison, in-place updating needs to disable sensing only for the duration of atomic updates. The third column in Table II lists the number of patch locations for each update case. The size of the atomic update set for *In-place* is equal to the number of patches and for *Trampolines*, it is always one since only a single write is needed to the base address of the jump target table.

Figure 5 shows the energy required to write the atomic update set to code memory on our reference device for each test case using the *In-place* approach. At most, the energy required is 323 nJ, which is more than two orders of magnitude smaller than the 89 uJ required to reboot TinyOS on the device. *Trampolines* has no down time since the change over happens instantaneously with one write.

C. Total Code Update Time

The total energy required for a code update is the sum of code transfer energy and code write energy. The former is proportional to the delta script's script, and the latter to the number of bytes written to code memory.

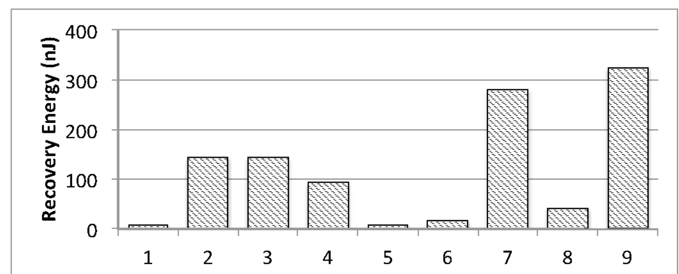

Fig. 5: Energy to recover from device down time for *In-place*.

Figure 6 shows the size of the delta script in bytes for each approach normalized to *RSync*. For simple changes like *Case-1*, where a constant value is modified, there is little difference between approaches. For more complex changes like *Case-3* where additional code is inserted, there is a much bigger difference. For this case, most of the delta script sent for *RSync* was modified call targets due to code shifting, not the actual inserted code. *Zephyr* was able to reduce the script to 45% on average compared to *RSync* by maintaining call targets with a function indirection table. However, it still needs to update the function indirection table itself. *In-place*. *Trampolines* has no shifting so the delta script has only patches. Hence, the script size is reduced to 21% on average compared to *RSync*.

Figure 7 shows the number of bytes written to code memory by the delta script for each approach normalized to *RSync*. *RSync* and *Zephyr* rewrite the entire image whereas *In-place* and *Trampolines* rewrite only the patched code. Hence, the cost to apply the update is typically less than 1% that of *RSync*. *Case-7* is an exception where the cost for *In-place* is 3.5% and for *Trampolines* is 5.3%. In this case, the size of patched code was a significant portion of the code image.

Update cost for *Trampolines* is slightly higher than *In-place* due to the need to setup the original and patched jump target tables and manipulate the trampolines. Update cost for *Zephyr* is sometimes slightly higher than *RSync* due to the extra writes to the function indirection table.

Figure 8 shows the energy required to perform code update

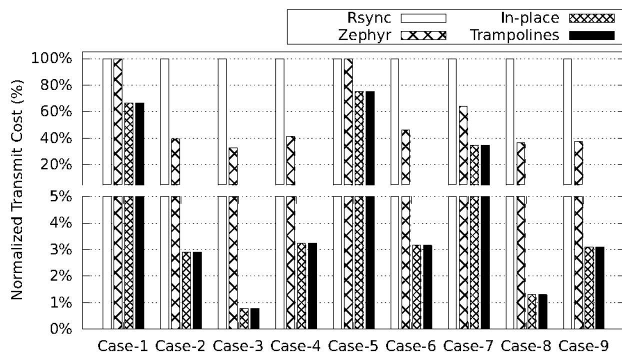

Fig. 6: Normalized size of delta script in bytes.

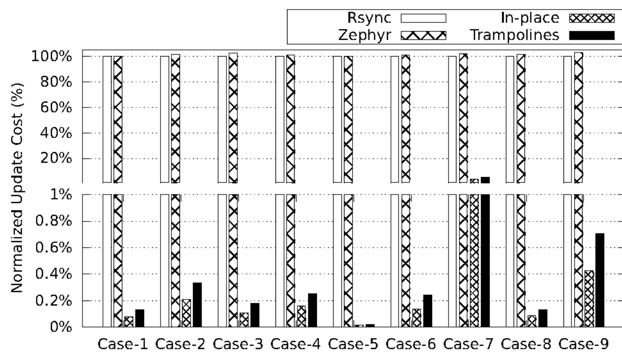

Fig. 7: Normalized number of bytes written to code memory.

for each approach broken down into code write energy, code transfer energy, and reboot energy. Note that while *Zephyr* reduces transfer energy significantly, it does not help with write energy for reboot energy since it still needs to write then entire code image to memory and also reboot. Our *In-place* and *Trampolines* eliminates or drastically reduces energy requirements for all categories.

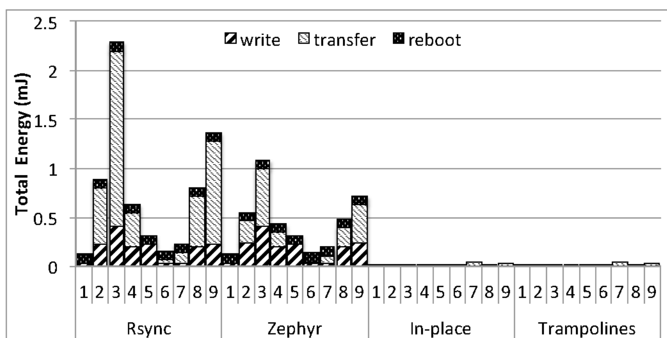

Fig. 8: Write, transfer and reboot energy.

D. Runtime Performance Cost

In-place patching can potentially negatively impact performance due to extra jumps to and from patched code, which would be unnecessary for conventional image rewriting (see Figure 1). However, in practice, both of our proposed approaches had negligible performance degradation, except in *Case-3*. The patch in *Case-3* was to OS timer polling code that is executed very frequently in a tight loop. Nevertheless, even in this code the overhead for *In-place* and *Trampolines* was only 2.7% in dynamic instructions.

VI. Conclusions

This paper describes in-place code updating for IoT devices that applies patches directly to the code image as it executes. This approach improves QoS by eliminating the need for reboots on code updates. It also shortens the turn-around time between generation of a code patch and application of the patch. Applying patches directly on the live code does away of the need to keep the old image around while constructing the new image, greatly decreasing the memory requirements of the device. Our strategy uses a trampoline to atomically update code with a single memory write. Results show a 78% reduction in delivery cost and 99% improvement in delta application cost. The performance overhead of in-place code updating is negligible.

VII. Acknowledgments

This material is based upon work supported by the National Science Foundation under grant numbers CCF-1422331, CNS-1012070, CCF-1535755, and CCF-1617071.

References

[1] “NAND Flash 101: An Introduction to NAND Flash and How to Design It In to Your Next Product,” https://www.micron.com/~/media/documents/products/technical-note/nand-flash/tn2919_nand_101.pdf.

[2] R. K. Panta, S. Bagchi, and S. P. Midkiff, “Zephyr: Efficient Incremental Reprogramming of Sensor Nodes Using Function Call Indirections and Difference Computation,” in *USENIX Annual Technical Conference*, June 2009.

[3] R. K. Panta and S. Bagchi, “Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks,” in *International Conference on Computer Communications (INFOCOM)*, April 2009.

[4] W. Dong, Y. Liu, C. Chen, J. Bu, C. Huang, and Z. Zhao, “R2: Incremental Reprogramming Using Relocatable Code in Networked Embedded Systems,” *IEEE Transactions on Computers*, Sept 2013.

[5] W. Dong, C. Chen, J. Bu, and W. Liu, “Optimizing Relocatable Code for Efficient Software Update in Networked Embedded Systems,” *ACM Transactions on Sensor Networks*, July 2014.

[6] W. Li, Y. Zhang, J. Yang, and J. Zheng, “UCC: Update-conscious Compilation for Energy Efficiency in Wireless Sensor Networks,” in *Conference on Programming Language Design and Implementation (PLDI)*, June 2007.

[7] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, “Fidelity and Yield in a Volcano Monitoring Sensor Network,” in *Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[8] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr, “Surviving Sensor Network Software Faults,” in *Symposium on Operating Systems Principles (SOSP)*, October 2009.

[9] J. W. Hui and D. Culler, “The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale,” in *Conference on Embedded Networked Sensor Systems (SenSys)*, November 2004.

[10] R. K. Panta, I. Khalil, and S. Bagchi, “Stream: Low Overhead Wireless Reprogramming for Sensor Networks,” in *International Conference on Computer Communications (INFOCOM)*, May 2007.

[11] P. B. Kessler, “Fast breakpoints: design and implementation,” in *Conference on Programming Language Design and Implementation (PLDI)*, June 1990.

[12] A. Tridgell, “Efficient Algorithms for Sorting and Synchronization,” 2000.

[13] “Over-the-Air (OTA) Update With the MSP430FR57xx,” http://www.ti.com/lit/an/slaa511a/slaa511a.pdf.

[14] “AX5243: Advanced High Performance ASK and FSK Narrow-band Transceiver for 27 - 1050 MHz Range,” http://www.onsemi.com/pub_link/Collateral/AX5243-D.PDF.

[15] “Avrora Simulator,” http://compilers.cs.ucla.edu/avrora/.