IP-SOC 2015

INNOVATIVE ENERGY SAVINGS USING GZIP IP WITHIN IOT DEVICES

Nikos Zervas, CAST, Inc.

Woodcliff Lake, New Jersey, USA

Abstract :

In this paper we look at how IP cores for hardware GZIP/Deflate based compression and decompression can significantly reduce power consumption in large categories of IoT devices. We will further show through multiple examples that the power reductions to be gained far exceed the active and idle power usage of the additional compression and decompression cores.

INNOVATIVE ENERGY SAVINGS USING GZIP IP WITHIN IOT DEVICES

The phrase "IoT" for Internet of Things has exploded to cover a wide range of different applications and diverse devices with very different requirements. Most observers, however, would agree that low energy consumption is a key element for IoT, as many of these devices must run on batteries or harvest energy from the environment.

Looking at how IoT devices actually use energy, it is clear that most:

1. Sit idle much of the time,
2. Wake up periodically or in response to an event,
3. Perform some kind of processing,
4. Transmit the results, and
5. Go back to sleep.

Most power-saving design strategies have focused on Step 3, making processing as efficient as possible. But new IP cores that simplify the incorporation of lossless data compression enable significant power saving in two additional phases, Step 2, booting or waking up, and Step 4, transmitting data. In the following sections, we will examine the opportunities for power savings in these two steps.

FIRMWARE COMPRESSION FOR LOWER ENERGY AND FASTER BOOT

Specifically, first we will show how GZIP data compression can help lower energy dissipation in embedded systems that use code shadowing, a common technique employed in IoT devices.

The basic idea is simple: on-the-fly decompression of previously compressed firmware reduces the data load and minimizes the number of accesses to long-term storage during boot or wake-up, hence reducing the energy (and the delay) during this critical phase of operation.

The possible energy and time-to-boot savings are proportional to the data compression level, which in turn depends on the compression algorithm and the code itself. Real-life examples we will explore here indicate that code size (and therefore power and time-to-boot) can be reduced as much as 50% using commercially available IP cores for hardware Deflate/GUNZIP decompression.

Furthermore, we will see that the savings much more than offset the extra resources used by building the right decompression core into the system.

Code Shadowing Versus Execute In Place

While they are in sleep mode, low-power embedded systems typically store their application code—and in some cases also application data—in a Non-Volatile Memory (NVM) device such as Flash, EPROM, or OTP.

When such systems wake up to perform their task, they run the application code via either of two methods:

- They fetch and execute the code directly from the NVM, called XIP for eXecute In Place, or
- They first copy the code to an on-chip SRAM unit, called the shadow memory, and execute it from there.

Which method is best depends on the access speed and access energy of the NVM memory. In general, NVM memories are significantly slower than on-chip SRAMs, and the energy cost of reading data from an NVM

memory is much higher than reading the same data from an on-chip SRAM (especially when data are accessed in random order).

While using shadow memory seems best when considering the system's active mode, the picture changes when we recall that an IoT device is usually asleep for most of its lifetime. Large on-chip SRAMs unfortunately suffer from leakage currents, and hence consume power even when in sleep mode, while most NVMs do not.

Designers therefore often chose code shadowing in cases where the shadow SRAM can be kept relatively small, or where the stringent real-time requirements make the slow access times of XIP unacceptable.

Lower-Power Code Shadowing via Fast Data Compression

We can address both issues—and steer the design decision towards the energy-saving code shadowing method—by reducing the size of the application code stored in the NVM.

Compressing the code using a lossless algorithm such as GZIP achieves this, but means the code must be decompressed before execution. Figure 1 illustrates an example IoT system architecture that does this. Here the NVM controller connects to the SoC bus (and from there to the on-chips SRAM) via a decompression engine, like the GUNZIP IP core offered by CAST [i].

Storing compressed code means fewer energy-expensive NVM accesses are required for system wake-up, but now we have added the extra step of decompression with its own delays and energy consumption. Whether this is an overall good idea depends on:

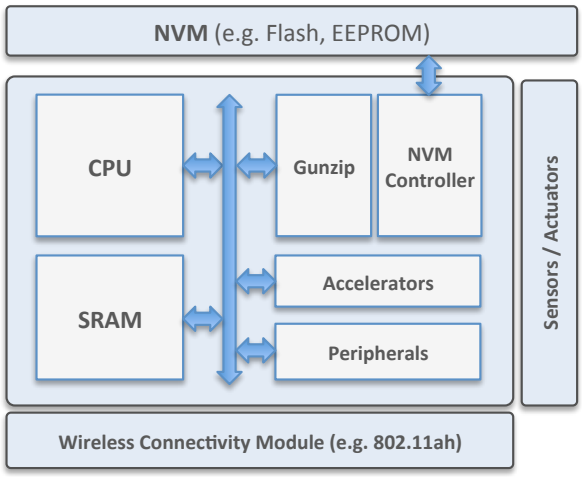

Figure 1: System architecture for Code Shadowing with Decompression.

A. How much can we reduce the size of the application code, i.e., what is the achievable compression ratio, and
B. What are the silicon area, power and latency requirements for the decompression hardware when we tune the compression algorithm settings to achieve a worthwhile compression ratio?

Let us next explore these factors by working through the numbers to see if the net energy savings of code shadowing using compression really provides a net energy savings.

Example Systems: How Much Energy is Really Saved?

Let us consider three IoT-like systems:

- In our first example, the R8051XC2 8051 [ii] microcontroller runs the Cygnal FreeRTOS port [iii].
- In our second example, the BA22-DE [iv] processor runs a sensor control application with multiple threads managed by a FreeRTOS port.
- In our third example, a Cortex-M3 processor [v] is running InterNiche Technologies' demo of embedded TCP/IP and HTTP stacks [vi].

In all cases, we use the ZipAccel-D GUNZIP IP core [1] to decompress the firmware as it is read out of a low-power serial Flash NVM memory.

The energy savings depends on the compression level, which in turn depends on the compressibility of the code itself (a function of the ISA and the application) and the chosen GZIP parameters. The GZIP parameters most affecting the compression are the type of the Huffman engine and the size of the History. These parameters also affect the silicon requirements and latency of our GZIP engine.

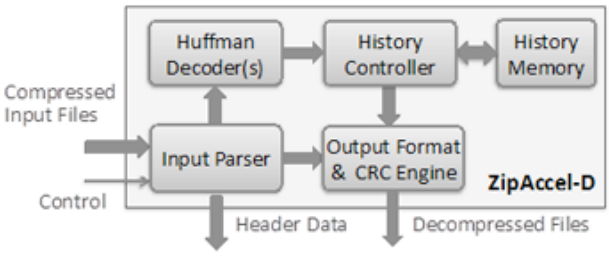

Figure 2: Hardware lossless data decompression engine; Huffman decoding type and History size are parameterized.

The uncompressed code sizes for our applications are 25.5 Kbytes for the 8051 system, 161 KBytes for the BA2 system, and 985 KBytes for the Cortex-M3 system. **Figure 3** shows the compression ratio for each of our example binaries and **Table 1** the area and latency of our decompression core for different sets of GZIP parameters.

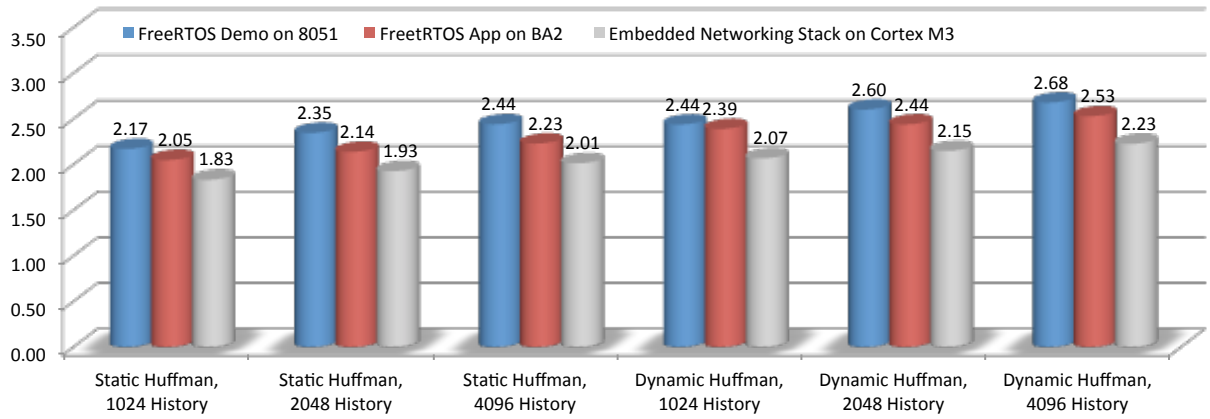

Figure 3: Compression Ratio for the image of InterNiche's demo of embTCP and embHTTP.

ZipAccel-D Configuration	Area in kGates	Memory in kBytes	Latency in clock cycles
Static Huffman, 1024 History	22	1.5	20
Dynamic Huffman, 1024 History	38	6.0	~1,500
Static Huffman, 2048 History	22	2.5	20
Dynamic Huffman, 2048 History	38	7.0	~1,500
Static Huffman, 4096 History	22	4.5	20
Dynamic Huffman, 4096 History	38	9.0	~1,500

Table 1: ZipAccel-D decompression core silicon resources and latency.

To keep the GZIP processing latency and silicon overhead low, we will use Static Huffman tables and a 2048 History. This makes our compressed code about half the size of uncompressed code, and similarly cuts the NVM size for code storage as well as the time and energy required to read code out during boot or wake up. **Table 2** and **Table 3** summarize these savings, assuming modern low-power serial Flash NVMs with a 5mA read current, 1.8V voltage supply and a 50MHZ read clock.

	Code Size in kBytes			Required NVM Size		
	System #1 (8051)	System #2 (BA2)	System #3 (ARM)	System #1 (8051)	System #2 (BA2)	System #3 (ARM)
Uncompressed Code	25.5	161	985	256kbits	2Mbit	8Mbit
Compressed Code	10.9	76	511	128kbits	1Mbit	4Mbits
Savings	**57.25%**	**52.80%**	**48.12%**	**50.00%**	**50.00%**	**50.00%**

Table 2: NVM Size Savings achieved using Code Compression.

	Boot Time in msec			Boot Power in mJ		
	System #1 (8051)	System #2 (BA2)	System #3 (ARM)	System #1 (8051)	System #2 (BA2)	System #3 (ARM)
Uncompressed Code	3.98	25	154	0.04	0.23	1.39
Compressed Code	1.7	12	80	0.02	0.11	0.72
Savings	**57.29%**	**52.00%**	**48.05%**	**57.25%**	**52.80%**	**48.12%**

Table 3: Boot Time and Energy Savings achieved using Code Compression, assuming an 1.8V Serial Flash, with 5mA read current and 50MHz read clock

The resource savings average about 50% and are clearly significant, but at what cost?

Analyzing Compression Overheads

Using compression in the manner described introduces overheads in two areas: time and energy. The ZipAccel-D decompression core we're using in our example systems [i] introduces a latency of 25 to 2000 clock cycles depending on whether static or dynamic Huffman tables are used for compression.

Even at 2000 cycles latency, and assuming that the decompression core would operate at the NVM's 50MHz clock, the additional delay added by the decompression core is just is 0.04msec. So, the additional delay due to compression is practically negligible, since the time to just read out the code from the NVM is two orders of magnitude higher.

On the energy side, the power usage of the decompression core is negligible while the system is active, but it also consumes energy while the system is idle. The significance of this idle-state power drain depends on the duty cycle of the system.

In our example systems, the idle power usage of the decompression core is 3 to 6 orders of magnitude lower than the power savings it enables. However, since energy is power over time, longer system sleep times make this extra power drain more important.

With the huge power savings we achieve, it is clear that storing compressed code and decompressing it when needed yields a net system energy savings for most IoT systems, even those with a duty cycle as low as a few msec per day.

TRANSMITTED DATA COMPRESSION TO REDUCE POWER CONSUMPTION IN CONNECTED DEVICES

Our second case in which data compression can save significant energy is Step 4 from our introduction, "Transmit the results."

Regardless of the specific network topology, bit rates, and communication standards, a significant amount of the energy budget of IoT nodes is consumed for communicating data. This is especially true for devices that use wireless communication links. In these, the radio frequency (RF) subsystem is the dominant energy consumer, and putting this power-hungry communication link to sleep for longer periods is essential for lowering overall power consumption.

The RF subsystem's active time—and therefore the energy consumed for data exchange—is more or less proportional to the amount of data that must be transmitted or received. So, using compression to reduce the size of that data can be an effective way to reduce the overall energy consumption of wirelessly connected IoT nodes.

Here we will describe an IoT device architecture that exploits lossless data compression, then try to quantify the energy savings this approach makes possible.

The Architectural Template

Most IoT nodes are controlled by a microcontroller consisting of a typically low-power embedded processor (e.g. 8051, ARM Cortex, or BA2x); peripherals for communicating with some sensor (e.g., for thermal, image, chemical, or mechanical readings or analog-digital conversion) or actuator (e.g., a step motor or electric valve); and possibly some accelerators for the computational intensive tasks (e.g., DSP functions such as filtering, or baseband modulation demodulation).

The typical IoT node stores firmware in an NVM memory, and communicate via an RF subsystem (e.g., an RF transceiver for Zigbee or 802.11ah).

How can introducing a compression accelerator to this generic architecture reduce its energy consumption?

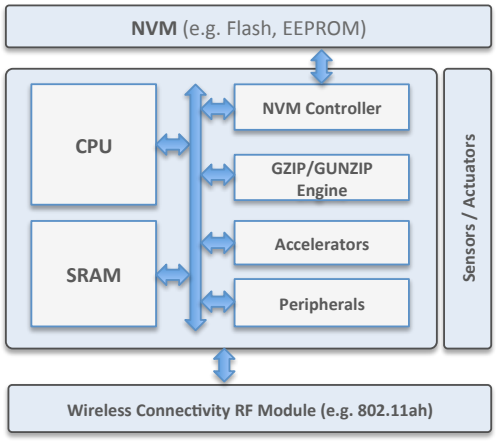

Figure 4: IoT architecture with integrated hardware data compression.

Here we are considering a hardware accelerator for compression. Compression algorithms are typically computationally complex, so implementing them in software requires a significant amount of code and may actually exhaust the processing bandwidth of a typical low-power IoT embedded processor. Moreover, a hardware compression engine can perform compression faster and at a much lower energy cost. We will use the same GZIP IP core in our analysis.

Analyzing the Potential Energy Savings

Achieving worthwhile levels of compression—and therefore, significant energy savings—depends on the type of data a system must transmit.

IoT devices that exchange large volumes of image, video, or audio data, are poor candidates for further savings since these typically already use compressed data.

This leaves a large group of IoT devices with lower data rates that:

a) Exchange a set of measurements (e.g. vehicle location coordinates, air pollution or meteoroidal measurements, liquids or chemical levels or level variations in a tank, a building's alarm system status, or a person's health monitoring data),
b) Are able to receive commands (e.g. "turn on heating in apartment number 5 – start in 10 minutes and keep max temperature below 70°F), and
c) May allow remote access via an HTML page.

From a compression point of view, this type of data resembles the characteristics of the simple text files, spreadsheets, and web files that office workers and engineers compress every day. It is typically highly compressible, with compression ratios in the range of 3:1 from lossless compression algorithms such as GZIP being common (see several examples in [vii]).

With less data to transmit after compression, the active time of the RF subsystem in our IoT device will be shorter. How much energy can this save? Consider:

- Typical low-power RF subsystems consume 10-30mA while active, and a few uA while asleep [viii}.
- A typical low-power IoT device microcontroller managing the RF subsystem will consume a few hundreds of uA when active and less than 1 uA while asleep.
- A GZIP accelerator configured to operate at 1Mbps consumes a few tens of uW while active and practically nothing in sleep mode (assuming the supply voltage is off).

Table 4 summarizes this power consumption information, looking at several sample low-power RF transceivers as well as low-power microcontrollers and the GZIP accelerator core available from CAST.

	LPRF Examples		**MCU Examples**		**Compressor**
	TI CC2500 (2.5GHz)	SemTech SX1272 (860MHz-1GHz)	TI MPS430 (MSP430F22374)	ST STM32L051C6 (@32MHz)	ZipAccel-C (65nm, 1Mbps)
Voltage (V)	2	2	3	2	1.2
Acrtive Power (mW)	42	32	1.17	8.90	0.18
Active Current (mA)	21	16	0.39	4.45	0.15
Sleep Power (uW)*	1.8	2	0.30	0.54	0.00
Sleep Current (uA)**	0.9	1	0.10	0.27	0.00

* WOR for CC2500, Sleep mode for SX1272
** Assuming Voltage cut-off and 25°C: LMP4 mode at 25°C for MSP430, stanby mode ar 25°C for STM32. 25°C for ZipAccel-C

Table 4: Power Consumption of Low Power RF Transceivers (LPRFs), MCUs, and a GZIP Compression Accelerator

We can apply this sample power data to our architectural template to estimate the potential energy savings compression makes possible. For this, we will assume an RF transceiver with 30mW active and 2 mW idle power, an MCU with 2mW active and 0.3uW Idle power, and a GZIP accelerator with the characteristics of Table 1.

The consumed energy (power over time) will depend on the device duty cycle (i.e., the percentage of time the device is active) and obviously on the compression ratio. **Table 5** shows the power usage for our example IoT device with and without compression, under the quite conservative assumption of a 2:1 compression ratio.

Example IoT node without compression				
Duty Cycle (Active sec per hour)	10	1	0.1	0.01
RF Active (Tx) Energy (mJ/hour)	300.00	30.00	3.00	0.3
RF Idle Energy (mJ/hour)	7.18	7.20	7.20	7.20
MCU Active Energy (mJ/hour)	20.00	2.00	0.20	0.02

MCU Idle Energy (mJ/hour)	1.08	1.08	1.08	1.08
Total Energy (mJ/hour)	**328.257**	**40.278**	**11.480**	**8.600**
Example IoT node with compression (Compression Ratio: 2:1)				
Duty Cycle (sec per hour)	5.000	0.500	0.050	0.005
RF Active (Tx) Energy (mJ/hour)	150.00	15.00	1.50	0.15
RF Idle Energy (mJ/hour)	7.19	7.20	7.20	7.20
MCU Active Energy (mJ/hour)	10.00	1.00	0.10	0.01
MCU Idel Energy	1.08	1.08	1.08	1.08
GZIP Active Energy (mJ/hour)	0.91	0.09	0.01	0.00
GZIP Idle Energy (mJ/hour)	0.00	0.00	0.00	0.00
Total (mJ/hour)	**169.175**	**24.370**	**9.889**	**8.441**
Net Savings (mJ/hour)	**159.082**	**15.908**	**1.591**	**0.159**
% Energy Saved	**48.5%**	**39.5%**	**13.9%**	**1.8%**

Table 5: Energy savings from Compression on a sample IoT node.

From this we can see that significant energy savings can be gained by compressing the data to be transmitted even in cases where the IoT device duty cycle is as low as 1sec/hour (which means the device is active 0.028% of the time). As we previously discussed, the overheads imposed by adding compression tot eh system are far outweighed by these savings.

Conclusions

We have seen that the negligible additional delay or energy usage from adding lossless data compression within an IoT system is far outweighed by power savings in at least two areas: firmware compression and transmission data.

For IoT devices that employ code shadowing, the compressed application code needs a smaller NVM device for long-term storage, and the system consumes significantly less time and energy reading the compressed code from the NVM into the on-chip SRAM.

For data exchange over a wireless network, compression clearly helps reduce the active time of the RF transceiver, which typically consumes relatively large amounts of energy while active. The energy savings will depend on the compression levels achieved (which in turn depend on compression algorithm), the exact power consumption characteristics of the device, and its duty cycle. However, using typical characterization data and rather pessimistic compression levels, we have seen that adding a compression accelerator can result to significant energy savings, even for devices with duty cycles bellow 0.1% (or 1 sec per hour).

---

i ZipAccel-D GUNZIP/ZLIB/Inflate Data Decompression Core: http://www.cast-inc.com/ip-cores/data/zipaccel-d/index.html

ii R8051XC2 High-Performance, Configurable, 8051-Compatible, 8-bit Microcontroller: http://www.cast-inc.com/ip-cores/8051s/r8051xc2/index.html

iii FreeRTOS Cygnal (Silicon Labs) 8051 Port: http://www.freertos.org/portcygn.html

iv BA22-DE 32-bit Deeply Embedded Processor: http://www.cast-inc.com/ip-cores/processors32bit/ba22-de/index.html

v ARM® Cortex ®-M3 Processor: http://www.arm.com/products/processors/cortex-m/cortex-m3.php

vi InterNiche Technologies embedded TCP/IP stacks demo: http://www.iniche.com/source-code/networking-stack/nichestack.php

[vii] The Canterbury Corpus Web page (http://corpus.canterbury.ac.nz/)

[viii] Designer's Guide to LPRG, Texas Instruments (http://www.ti.com/lit/sg/slya020a/slya020a.pdf)