Optimizing for Energy in High-Level Programming Languages on Embedded Devices

Péter Henrik Gombos

Master of Science in Computer Science
Submission date: June 2015
Supervisor: Gunnar Tufte, IDI
Co-supervisor: Stefano Nichele, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

The use of embedded systems has exploded recently, and thus also the number of developers for embedded systems. But the traditional way of programming embedded computers is hard and error prone, and the use of high-level programming languages is preferred. Unfortunately, with high-level languages come a high level of power usage.

This thesis examines techniques of bringing high-level programming languages, specifically JavaScript, to embedded computer systems, with a focus on driving down the energy use. Three different JavaScript engines, Tessel, Espruino, and io.js, are tested on the Tessel hardware and the Raspberry Pi. The tests consists of trivial operations run in a loop repeatedly.

io.js is shown to execute the programs the fastest of the three engines, and also uses the least amount of power per iteration. The Tessel is shown to have a sub-optimal implementation of some operations, which limits its performance. Espruino runs the benchmarks poorly, as they were not written with the execution model of Espruino in mind.

It is concluded that by increasing the execution speed of the software platforms, the system will be more power efficient.

Sammendrag

Bruken av integrerte datasystemer har eksplodert de siste årene, og dermed også antallet utviklere for disse systemene. Men den tradisjonelle måten å programmere integrerte systemer på er vanskelig og full av feilskjær, og bruk av høynivå programmeringsspråk er å foretrekke. Uheldigvis kommer høynivåspråk med et høyere energiforbruk.

Denne oppgaven undersøker teknikker for å bringe høynivå programmeringsspråk, spesifikt JavaScript, til integrerte datasystemer, med et fokus på å begrense energyforbruket. Tre forskjellige JavaScriptimplementasjoner, Tessel, Espruino og io.js, blir testet på to hardwareplattformer, Tessel og Raspberry Pi. Testene består av trivielle operasjoner i en løkke reptertet et stort antall ganger.

io.js kjører programmene raskest, og bruker også minst energi per iterasjon. Ytelsen til Tessel på noen operasjoner begrenses av suboptimal implementasjon av disse operasjonene. Siden programmene testet er skrevet i standard JavaScript, kjører ikke Espruino disse effektivt. Espruino kjører programmer på en spesiell måte, og krever at kode er skrevet for denne modellen for å være effektiv.

Konklusjonen er at ved å øke kjørehastigheten til softwareplattformene vil systemet bli mere energieffektivt.

Preface

This work is the report for my Master's project at Norwegian University of Science and Technology (NTNU). The project was performed during the spring semester of 2015 at the Department of Computer and Information Science. Associate Professor Gunnar Tufte has been the supervisor of the project.

I assume the reader has some programming knowledge, but it should be possible to follow the project without any prior knowledge.

Acknowledgement

I would like to thank my parents, whom without I truly could not have completed this project. Also, I want to thank Gunnar Tufte, his input on both the project and this thesis has been invaluable.

Cover image is by Jay Hilgert.

Péter Henrik Haldorsen Gombos

Contents

1 Introduction

Energy consumption in embedded computer systems is a topic of increased importance, with the explosion of their use in recent years. This rise is driven by the concept of the *Internet of Things* (IoT). IoT means that some physical device is connected to the Internet, allowing it to be remotely controlled and have functionality that an isolated system can not have. (Kopetz, 2011) Embedded systems allow the development process to tailor both hardware and software to the specific use of the system, making the embedded system not a general programmable computer. Despite the advantages this gives for the application, such as specific optimizations, this makes it harder to develop a system, as there is no common platform. Additionally, an embedded system designer needs to be knowledgeable in both the software and hardware domain, and developing both will add time to the development of the device.

A solution to these problems is to use more generalized hardware, and use software abstractions to ease the development. But this gives away a lot of the low-level control over the device, losing the possibilities for many optimizations, and adds overhead to the execution process. As an attempt to consolidate these two competing impulses when developing an embedded application, this work will try to investigate the energy consumption of using high-level programming languages on embedded devices. Specifically, the use of JavaScript as the development language on different software and hardware platforms is investigated.

1.1 Embedded devices

So what exactly is an embedded computer system? In Wolf (2008), it is defined as “any device that includes a programmable computer but is not itself intended to be a general-purpose computer”. Grimm et al. (2013) says they are “the integration of a (microelectronic) system and its software into a larger, often autonomous, system that often monitors and/or controls equipment without the need for manual intervention.” From these definitions, it can be understood that an embedded system is a programmable computer that interacts in some way with the physical world in a device-specific application. Embedded systems range from ABS brakes in a car to cell phones.

When designing an embedded computer system, the resources can be individually tailored to the application of the system. For example, using Bluetooth is popular for communicating with embedded systems, but if the application does not require wireless communication, there is no need to add Bluetooth capabilities to the design. This tailoring applies to every part of the system, requiring the designer of the system to balance contradicting constraints and needs. In a battery driven system, a powerful battery that lasts a long time would be optimal, but the size of the battery will limit the systems’ usability. For example, the battery used by the Tesla Model S can power the car for almost 430 km. A similar battery in a cell phone would last for a long time, but as the battery weighs nearly 600 kg, it would be impractical. (2013 Tesla Model S Review, Car and Driver) Other constraints imposed on the systems can be memory size, storage space, maximum operating temperature and deadlines for delivering data.

Power consumption is a constraint that is especially important in embedded systems, in a way that is not relevant to some other computer systems, like desktop machines. A battery powered device needs to keep functioning for the longest time possible on a single charge. But many systems cannot be charged at regular intervals, for example sensor systems in areas that are hard to reach, such as at the bottom of the sea or embedded in concrete. These systems need to be able to function for a long time on a single battery. As lower energy use also means lower heat dissipation, better energy efficiency will lead to a cooler system, which is crucial for hand-held devices like cell phones.

1.2 Compilation and interpretation

When computers first appeared, they were programmed manually with binary code, which is time-consuming and error prone to write, and requires the programmer to remember various arbitrary codes. Instead, assembly languages were developed, the first already in 1949 with further developments through the 1950s, which provides a one-to-one mapping of easier to remember function names to the machine code. (Salomon, 1992) While far better for the programmer, assembly programming is still very time consuming and difficult to write, and when creating a complex application, the data models the assembler provides are too basic. Virtually all programs written today are written in a high-level programming language, requiring translation of the programs to a format that the machine can run. (Aho et al., 2006) This translation can be done in two different ways, compilation or interpretation.

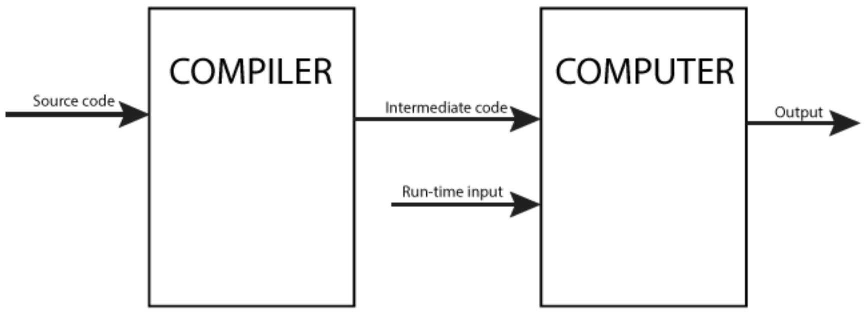


Figure 1.1: A compiler producing intermediate code, and a computer executing the program

Compilers take source code in a given language as input and outputs code in another form either into a lower level language or into another programming language entirely. Compiling to another high-level language is known as transcompiling, or simply *transpiling*. The compiled program when run takes the input and produces an output. Figure 1.1 illustrates how a compiled program runs on a computer. An interpreter, on the other hand, is a program running the code while at the same time taking any input, and then produces the output, illustrated in fig. 1.2.

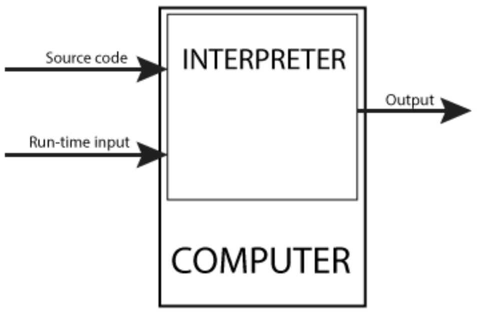


Figure 1.2: An interpreter executing a program on a computer

While compilation and interpretation give a lot of advantages over writing programs in assembly language, they do add some overhead while running programs. A compiler that translates anything but a trivial program can produce many equivalent and correct outputs. There are often many ways of implementing the abstractions of a high-level language in a lower language. Usually, there will most likely be added more instructions after a compiler pass compared to writing the assembly code by hand. In the case of interpretation, the interpreter running on the computer by definition adds extra instructions. On the other hand, adding automated passes of the code allows for some optimization techniques, which will be explored in chapter 2.

1.3 JavaScript

JavaScript is a dynamic, weakly typed language, created for allowing interactivity on web pages. Developed in 1995 by Netscape, it became the basis of the ECMAScript standard in 1997. (A Short History of JavaScript) This standardization of JavaScript allowed it to become the most used programming language on the web, by being available in every browser. Currently, almost 90% of web pages uses JavaScript. (W3Techs, a) Newer developments in the JavaScript world is Node.js, later forked into io.js, a standalone framework for running the V8 JavaScript engine, allowing for the use of JavaScript on the server side. More on the io.js framework and the V8 engine is presented in chapter 3. With being available in just the last three years, already 0.2% of all websites are currently using some server side JavaScript. (W3Techs, b)

This dominance of JavaScript on the web has created a large community around the language, with a lot of tools and available open source software. In NPM, the Node Package Manager, over 150,000 JavaScript packages are available for download.[1] As more people are using JavaScript, there is a drive to expand its areas of deployment and usage, seen through all the different projects with the aim of running JavaScript directly on hardware. Besides the benefits of being a high-level language, there is also the bonus for the developer not to have to learn a new programming language when developing for an embedded platform.

1.4 Problem

When creating embedded systems, there's often a need to use lower level languages to achieve the desired power and speed requirements. It can be difficult to program in these languages, with few abstractions and a hard to understand data-model. On the other hand, high-level languages offer many improvements for the developer, as well as a familiar programming environment. Unfortunately, high-level language often mean a high level of energy use. To consolidate the competing directions of embedded development, this thesis asks:

> "What are energy efficient ways of bringing high-level programming languages to embedded devices?"

This question will be explored by comparing the energy use of some JavaScript engines, particularly with the Tessel 1 and a Raspberry Pi board running Espruino and io.js while running benchmark code.

1.5 Related work

The first research into optimizing software to achieve better energy performance was done by Tiwari et al. (1994). Before their work, power measurement tools were only available at the circuit and logic level. They introduced a way of estimating the energy cost of the instructions that are available on the processor, by measuring the current flow into the microprocessor with an ampere meter. Also introduced was a model for

[1] `https://www.npmjs.com/`

estimating the power consumption of a program by looking at the instructions of the program. With this data, they could optimize code running on the processor by using a strategy that minimizes the use of power-hungry instructions.

Russell and Jacome (1998) found that the model developed by Tiwari et al. was needlessly detailed. Their model estimates power consumption of a program within 8% accuracy, by using the average power consumption per cycle of the processor, multiplied with the execution time in cycles of the program. Their conclusion is that adding optimizations that minimize execution time of a program, will also reduce the energy consumption of the same program.

Ortiz and Santiago (2008) looked at source code optimizations to find if they could lower the energy consumption. They found that the impact of the optimization techniques varied between the platforms used. This device specific optimization was explored further by De Lima et al. (2013), where the authors tried to find a set of compiler optimizations that give the best result for the given program. They did this by reducing the number of possible sets of optimizations, and then tested code compiled with the sets of optimizations, finding the set that provided the best performance gain or was most power efficient. While their research focused on a desktop computer, the same could be done for an embedded system.

Kavvadias et al. (2004) found that they could estimate the energy consumption of a program by using the base cost of each instruction. The model they developed shows that the energy use is dependent on the instruction energy use, the interinstruction costs of each instruction pair, and the costs of pipeline stalls.

$$PE = \sum_{1}^{n} E_i + \sum_{1}^{n-1} O_{i,i+1} + \sum \epsilon \tag{1.1}$$

Shown in eq. (1.1) is the mathematical model of their research, where E_i is the energy consumed during the execution of instruction i, $O_{i,j}$ is the interinstruction cost of instructions i and j, and ϵ is the cost of a pipeline stall. The model shows that reducing either the number of pipeline stalls or the number of instructions in the program will reduce the energy consumed by the program.

In other research done in the same field, Valluri and John (2001) found that compiler optimizations done for speed positively affected energy consumption. Specifically, reducing the number of instructions that the processor needs to execute, reduces energy consumption. Some optimizations that the research found favorable for energy use were loop unrolling and function inlining.

Fortuna et al. (2010) found promising results in trying to parallelize JavaScript programs running on the web. In the design, JavaScript is inherently single-threaded, which might lead to bottlenecks during execution. While the hardware platforms tested in this thesis does not support parallel execution, it might be possible to use some of the techniques in the paper to achieve better performance on future devices.

1.6 Outline of this thesis

Chapter 2 (page 9) goes deeper into what can be done to create an energy efficient embedded device. In chapter 3 (page 15) the method of the experiment is explained, together with the rationale behind the experiment. The results of the this experiment is presented in chapter 4 (page 25), and these are discussed in chapter 5 (page 35). Concluding remarks are added in chapter 6 (page 45).

2

Creating an Energy Efficient Platform

When creating an energy efficient computer, there are two main parts to focus on, the hardware and the software. Often they have to work together to create the best possible system. This chapter will explain different energy saving techniques, by exploring both the hardware and the software view.

2.1 Hardware

An easy way of saving energy is to let the computer run slower, executing fewer instructions in the same amount of time. As it is the act of executing an instruction that draws energy, running fewer instructions will lead to lower power use. But as demonstrated by fig. 2.1, if the execution time of a program on a slower computer is long enough, the total energy use may be higher than on a faster computer.

To exploit this behavior, dynamic scaling of the clock, as well as the input voltage, can be used. By letting the operating system gather data from performance- and energy-critical events, the operating system can manage the demands of each application. (Weissel and Bellosa, 2002) Later research has found that the gain from dynamic frequency scaling has diminished, as, for instance, sleep states in processors has become better. (Le Sueur and Heiser, 2010)

Sleep states, or low power states, is a term for states where less of the computer's resources are available. While waiting for input, for instance, the CPU does not need to be running, as it has no work to do. Doing so is called busy-wait, and consumes a lot of energy, because the execution is spending time in a loop where the only thing done

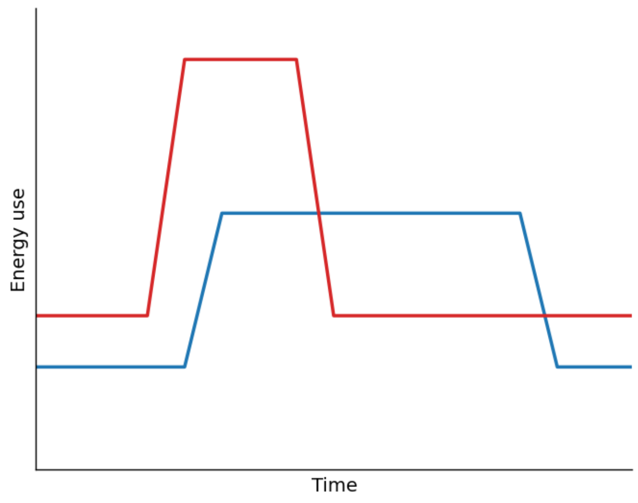


Figure 2.1: Energy use of a fast processor (red) against a slow processor (blue) running the same program

is checking if the computer has received any input. It is the computer equivalent of children on a road trip constantly asking "are we there yet?" Introducing *interrupts* is a better concept, where the application tells the operating system that it is waiting for an interrupt. The OS can then either schedule other programs to run or turn off resources that are not needed to wake up again, i.e. enter a sleep state. When the event that the program is waiting for happens, the OS fires an interrupt and gives control back to the program. In the program, a special function called *interrupt handler* runs, gathering the input. In input-heavy programs, where the time spent waiting for data dominates the calculation time, a lot of energy can be saved by using sleep states.

For example, the Silicon Labs EFM32 Giant Gecko has five energy modes with different hardware capabilities available in each mode, as shown in fig. 2.2. EM0 is the normal running mode, where the CPU is available. In figure 2.2 it is shown in light green. (EFM32 Giant Gecko Reference Manual, 2014) The EFM32GG990 uses typically 219 µA/MHz when operating in EM0, the normal running mode. When entering EM1, the first and lightest sleep mode, the typical value is 80 µA/MHz, and in the deepest sleep mode, EM4, it only consumes 0.02 µA. The use of these energy modes does have some drawbacks, namely that waking up from deeper sleep modes takes time. On the EFM32GG990 device, it takes 163µs to wake up from the deepest sleep mode, which

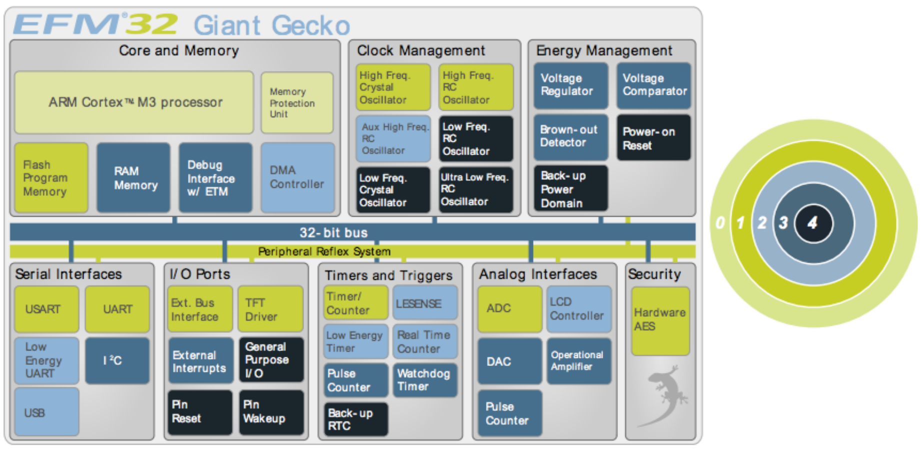


Figure 2.2: The energy modes of the Silicon Labs EFM32 Giant Gecko

might cause problems in timing sensitive systems. Also, it consumes more energy to wake up from a sleep state. (EFM32 Giant Gecko 990 Datasheet, 2014)

Sleep states taken further lead to turning off parts of the computer. Because of the failure of Dennard Scaling (faster transistors and more of them on a chip leads to progress) in recent years, the term *dark silicon* has been introduced. When more and more transistors are running on a single chip, the heat of the device becomes too high to sustain, and some parts of the chip must be turned off, or kept dark. This is a disadvantage when the goal is to use as much of the chip at the same time as possible. Research is currently being done into leveraging this property when designing the system, i.e. purposely turn off some parts of the chip that can be beneficial. (Taylor, 2012)

Kumar et al. (2003) created an architecture for multiple heterogeneous cores, using existing cores with a similar instruction set. By using an informed strategy for choosing the core to run the program on, and having the ability to switch core dynamically while running, they achieve an increase in energy efficiency. Similarly, the SHMAC (The Single-ISA Heterogeneous MAny-core Computer)[1] project tries to create a computer with multiple different cores. Here some cores are more capable of specific tasks than others, using accelerators, out-of-order processors or vector processors. The computer can then allocate a particular core for a task, and turn off the parts of the CPU that are not used. In the same way as Kumar et al. described, energy can be saved by the turn-

[1]`http://www.ntnu.edu/ime/eecs/shmac`

ing off of the machine while at the same time achieve a performance boost by using specialized hardware. (Jahre, 2013)

2.2 Software

Running optimized code on the platform might lead to lower power consumption, as shown by Kavvadias et al. (2004) and Valluri and John (2001). Optimizing the code leads to fewer instructions in the program, which leads to lower energy use for the entire program. When compiling a program, instead of naively translating each expression into instructions, the compiler can analyze the source code and optimize the use of instructions. Some of the common optimizations the compiler can do, are listed below. (Aho et al., 2006, chap. 9)

Global common subexpression

Find some subexpression in the code that is calculated multiple places, and instead only calculate it once, putting the result of the calculation into the places that redundantly calculated the expression.

Constant folding

Deducing that some variable or calculation is constant, and replacing it with this constant.

Dead code elimination

Finding that some part of the code is unreachable at run-time, and instead of having to add the unreachable code to the compiled code, eliminate it.

Function inlining

Move a function definition to the place where the function is called in the code, to remove the added work of calling a function.

Loop unrolling

Reduce the number of iterations in a loop by doing more work in each iteration, for example, do the calculations for originally three iterations in one.

Code-motion

Find code that is loop-invariant, i.e. calculations that does not need to be calculated for each iteration in the loop, and move it outside the loop body.

Reduction in strength

Swap an expensive instruction for a cheaper one, e.g. change a multiplication by two with a left shift.

The strength of these optimizations becomes apparent when the compiler performs several of them and multiple times. For instance, after a function inlining, some expression in the code that was former the function can be a constant, which could lead to some code in the function being dead. As shown by De Lima et al., the order of the techniques can impact the results of the optimization.

An interpreter executes the code as it runs, and it cannot get the insight a compiler does to perform optimizations. If the interpreter were to implement the same techniques, it would have to do two passes of the code, and the whole purpose of the interpreter would vanish. But there is a middle ground, called *just-in-time compilation*, or JIT compilation for short.

A JIT compiler runs as the program is executing, just like an interpreter. But it does not necessarily only translate one expression at a time, it can compile larger blocks, allowing for optimizations on these parts before execution. (Aycock, 2003)

Another way of making the software save energy is to not execute code at all. With the rise of Internet-connected devices, it has become feasible to load off the computation-heavy operations to some other computer. A typical example of this is making a server create queries and retrieve data from a database, which might be computation intensive. A newer example is in gaming, where a handheld device gets rendered 3D data from a dedicated gaming computer as a video stream and sends input signals back. The PlayStation Vita[2], made by Sony, is a small gaming device, that can play any PlayStation 4 game through a network connection, despite not being powerful enough to render the games.

The operating system on an embedded system uses energy. While strictly speaking not needed for writing assembly programs, when the computer runs high-level pro-

[2]`https://www.playstation.com/en-us/explore/psvita/`

grams, the operating system performs actions that is either required or convenient. But the more features the operating system has, the more actions are performed in the background, leading to higher energy use. Thus to save energy, optimizing the operating system can yield results. When designing an embedded platform, there are two main strategies to use. Either create a minimal framework that functions as an operating system, or use an existing system and optimize it for the device.

3 Method

To fulfill the goals of this thesis, an experiment comparing the energy use of different JavaScript engines was developed: Measuring the current drawn while executing benchmark programs in different JavaScript engines on various hardware platforms, and using these measurements to find the power usage of a single expression. By running the programs a large number of times, it is possible to obtain an average power consumption of the expressions. All of the programs are trivial operations in a loop, and by calculating the average consumption the energy cost of a single expression in the JavaScript engine can be found.

The experiment will look at three different software solutions to execute JavaScript code, each with its implementation tactic, namely Tessel, Espruino and io.js. The tests consist of running these software solutions on two different hardware platforms, the Tessel 1 and the Raspberry Pi 1 Model B.

The goal of the experiment is to look at differences in the energy use of the different JavaScript implementations, which means that it is not the exact numbers that are of interest, but the trends.

3.1 Benchmarks

When testing the platforms, the benchmarks consisted of four programs, all following a similar pattern: Evaluating an expression in a loop repeated 1,000,000 times. The programs are named after the expression used in it, testing various JavaScript features; Addition, Multiplication, Closure, and Left shift. The addition program is shown in listing 3.1, and the multiplication program in listing 3.2, using basic mathematic operations.

```
for(var i = 0; i < 1000000; i++) {
    var a = 1;
    var b = 2;
    var c = a + b;
}
```

Listing 3.1: The Addition program

```
for(var i = 0; i < 1000000; i++) {
    var a = 1;
    var b = 2;
    var c = a * b;
}
```

Listing 3.2: The Multiplication program

In listing 3.3, the Left shift program is shown. Like the addition and multiplication program, it consists of a single expression, left shift. To left shift *a* by *b* is to add b zeros to the end of a's binary representation. For example:

$$1_{10} << 2_{10} = 1_2 << 10_2 = 100_2 = 4_{10}$$

```
for(var i = 0; i < 1000000; i++) {
    var a = 1;
    var b = 2;
    var c = a << b;
}
```

Listing 3.3: The Left shift program

```
for(var i = 0; i < 1000000; i++) {
    var a = 1;
    var c = (function () {return a})();
}
```

Listing 3.4: The Closure program

The Closure program, seen in listing 3.4, evaluates a *closure*, which accesses a global variable. A closure is a concept often used in JavaScript programs, where an unnamed local function is evaluated where it is defined. In this example, it may seem trivial, but as JavaScript only has a scope per function, closures are an important tool to not clutter the global scope. And with the single threaded nature of the language, as well as its higher order, closures are an important flow control concept in applications.

3.2 Hardware Platforms

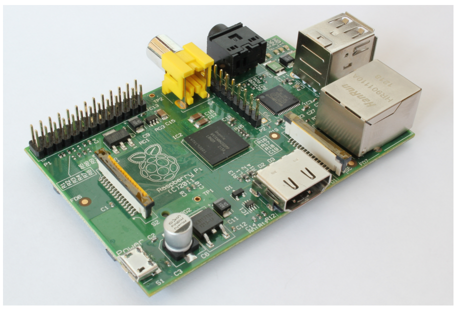


Figure 3.1: Raspberry Pi Model B, source: `https://commons.wikimedia.org/wiki/File:RaspberryPi.jpg`

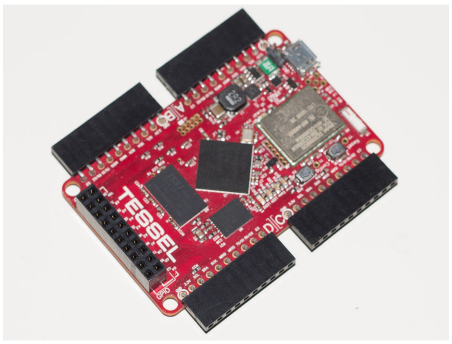


Figure 3.2: The Tessel 1

As can be gathered from table 3.1, the Raspberry Pi is a much faster machine than the Tessel, clocked at almost four times higher, and with 16 times the amount of RAM. This difference means that the Raspberry Pi can run larger programs and do it faster, but also that it will use more power.

	Raspberry Pi	Tessel
CPU	BCM2835	LPC1830
Core	ARM1176	ARM Cortex M3
Architecture	ARMv6	ARMv7
Clock Speed	700 MHz	180 MHz
RAM	512 MB	32 MB
Flash	SD-card	32 MB

Table 3.1: Comparing a Tessel 1 with a Raspberry Pi 1 Model B (Raspberry Pi FAQs; Tessel Harwdare Documentation)

Silicon Labs EFM32 Giant Gecko

When developing the experiment, another hardware platform was intended to be tested. The Giant Gecko from Silicon Labs is a microcontroller created for low energy use cases, described further in section 2.1. To develop applications for the platform, Silicon Labs sells development boards. In this project, the EFM32GG990-DK3750 was used for this purpose. It uses a CPU that is summarized in table 3.2. It is a much less powerful device than both the Tessel and the Raspberry Pi, clocked at a third of the Tessel, and containing a small fraction of the memory available on the other platforms.

CPU	GG990
Core	ARM Cortex M3
Architecture	ARMv7
Clock speed	48MHz
RAM	128KB KB
Flash	1024 KB

Table 3.2: Key data of the Giant Gecko 990 (EFM32 Giant Gecko 990 Datasheet, 2014)

3.3 Software Platforms

Tessel

Tessel[1], developed by Technical Machine, is a project created for offering software developers a platform for developing hardware applications. It consists of both a hardware platform, described above, and a software platform. This software platform includes a runtime for the hardware, interpreting the code; a command line interface to easily run programs, and a compiler. Instead of interpreting JavaScript, Tessel compiles it to Lua and interprets the generated Lua code on the device.

Lua is another programming language, designed to be fast and lightweight, and is easily embeddable. In many ways, it is similar to JavaScript, so a translation from JavaScript to Lua is not very difficult. There exists two main implementations of Lua, the official interpreter, and LuaJIT, a highly optimized just-in-time compilator, and virtual machine. (About Lua)

One of the rationales behind choosing Lua as the intermediate language was that Lua has a low memory profile and is easy to embed. The V8 engine was the only alternative at the design time of the Tessel. But at the time, Technical Machine did not think using V8 could be reconciled with the low power goals they had for the Tessel. To further drive down energy use, the LuaJIT compiler was introduced, first only using the virtual machine, but with the goal of enabling just-in-time compilation in the future. (A New Engine For Your Tessel, 2014)

The compiler of the Tessel, the Colony Compiler[2], is a simple lexical parser, that translates the JavaScript expressions into the Lua equivalents. Consequently, while a

[1]`https://tessel.io`
[2]`https://github.com/tessel/colony-compiler`

complete compilation of the source code is done, it does not do any optimizations to the output.

Recently, Technical Machine has shifted focus away from creating a truly low-power device. Ease of development is more important for the Tessel going forward than minimizing the power consumption. Therefore, the Tessel 2 will run io.js using a Linux-based operating system. (Moving Faster With io.js, 2015)

Espruino

Espruino[3] is another project that aims to bring hardware to software developers, also through JavaScript. But unlike Tessel, the goal is to target devices with memory as small as 128kB Flash and 8kB RAM. The Espruino project consists of both a software and a hardware platform, but to achieve the low memory goal, the software implementation includes a lot of trade-offs.

The Espruino virtual machine (VM) is an almost complete JavaScript implementation. The difference between Espruino and other JavaScript implementations is that it does not translate the source code to byte code but executes the source directly, one expression at a time. As the developers want to keep the JavaScript source code on the device while executing to be able to edit it on the device, it is not translated due to memory concerns. If the VM translated to intermediate bytecode, it would need twice the amount of storage for the same program to keep it all on the device.(Espruino Interpreter Internals)

Other optimizations done for memory minimization include using a linked list for storage of arrays and objects, allowing for constant time insertion and deletion. Also available are typed arrays which are a direct mapping of memory to the data structure, allowing write applications with a more optimized memory model. (Espruino Performance Notes)

io.js

io.js[4] is a framework created with the goal of using the V8 JavaScript engine without a browser, and with a rich input and output (I/O) API. The framework is asynchronous

[3] `http://www.espruino.com/`
[4] `https://iojs.org`

and event-driven, designed for creating scalable network applications. The io.js framework uses an event loop that runs callbacks, functions that are registered to be run when an arbitrary event happens. io.js is therefore different from traditional thread-based programming platforms, where I/O blocks further execution of the code. (About Node.js)

As a wrapper around V8, io.js also functions as an easy way of installing a JavaScript runtime on computers, allowing JavaScript to be an alternative to other programming languages.

io.js is a fork of Node.js, with a different development schedule and philosophy. Currently, io.js uses a newer version of the V8 engine than Node.js. But the io.js team has recently announced that they will be joining the new Node Foundation and renaming the project Node.js soon.[5] Thus the terminology in this thesis a bit hard, as io.js may suddenly have changed its name. But as the future Node.js will be more or less based upon the io.js project, the least confusing at the moment is to use io.js consistently.

V8

The V8 JavaScript engine is built by Google, and used in the Google Chrome Web Browser. It is developed with a focus on creating the fastest JavaScript engine possible and outperforms all other major JavaScript engines available today. There are three major design areas to achieve the speed required (V8 Design Elements):

Fast property access, as JavaScript is a dynamic language, property access usually is slow, due to it requiring a dictionary lookup. V8 uses dynamically created hidden classes to be able to access properties from offsets, requiring a single load instruction. It only requires these classes to be created once for each object, which leads to a faster creation of any other objects of the same class.

Dynamic machine code generation, the engine uses just-in-time compilation, directly to bytecode. Because of the hidden classes created for the fast property access, the compiler can guess that after property access, the current class will be used for all future accesses in the same section of code.

[5] `https://medium.com/node-js-javascript/io-js-week-of-may-15th-9ada45bd8a28`

Efficient garbage collection, V8 uses a strategy for memory reclaiming that aims at running for as short time as possible. To do this, it stops program execution when performing garbage collection and does not process the whole heap in most collection cycles.

Operating system

An operating system is needed to run programs on a computer, controlling the input and output mechanisms, and schedule which program is to run. The Tessel 1 firmware includes a basic OS, so a developer does not need to set up an OS.

The website of Raspberry Pi recommends the Raspbian Linux distribution, which is based on Debian 7. Since the Raspberry Pi has an ARMv6 processor, but Debian only supports ARMv7 natively; this distribution was created separately. But Raspbian is not actively maintained, and has become quite outdated. Debian is currently in version 8.

uClinux is used through the PTXdist build system to get an operating system on the Giant Gecko. PTXdist consists of a complete toolchain for building Linux, and a lot of rules to build packages for various systems, among them the Giant Gecko DK3750.[6] It is not a distribution, but an “executable documentation”, meaning that it contains the steps necessary to build the target system in scripts that can be read or executed. (Pengutronix, 2014)

3.4 Experimental setup

To power the experiment, the Keysight E3631A power supply was used. As both the hardware platforms were running experiments at the same time, they were both connected to the power supply, with a common ground. The voltage was set to 5.015V, as it is the recommended operating voltage for both the Raspberry Pi and the Tessel. (Power Supply, Raspberry Pi Foundation; Powering Tessel, Technical Machine) The multimeter, Keysight 34410A, was connected in series with the power supply and the device, set to ADC measurement. There was one multimeter for each device while running in parallel. Figure 3.3 shows a circuit diagram for this experiment. Both devices received power through a Micro USB cable. A cable was cut, and with clamps, the power and ground

[6]`http://git-public.pengutronix.de/?p=OSELAS.BSP-EnergyMicro-Gecko.git`

wires were connected to the power supply. (Universal Serial Bus Micro-USB Cables and Connectors Specification, 2007)

The multimeter supports logging over LAN, allowing to monitor the experiment remotely. Together with the Raspberry Pi's remote SSH-access, the experiments on the Pi could be remotely started and controlled. To fully automate the experiment on the Raspberry Pi, a shell script was written to restart the program when it had run its course. After each run, the device would sleep for 0.5 seconds before the reset, to tell the difference between each run.

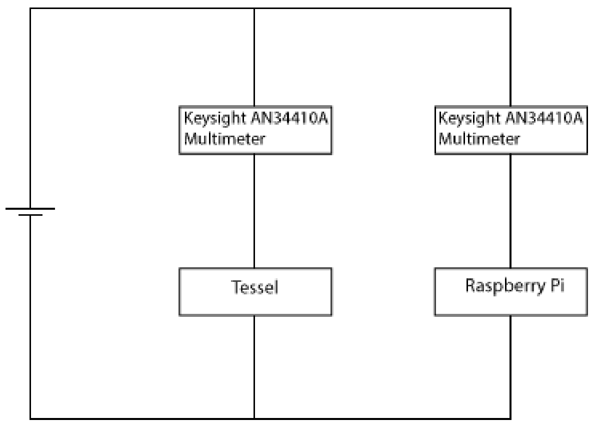


Figure 3.3: Circuit diagram of the experiment

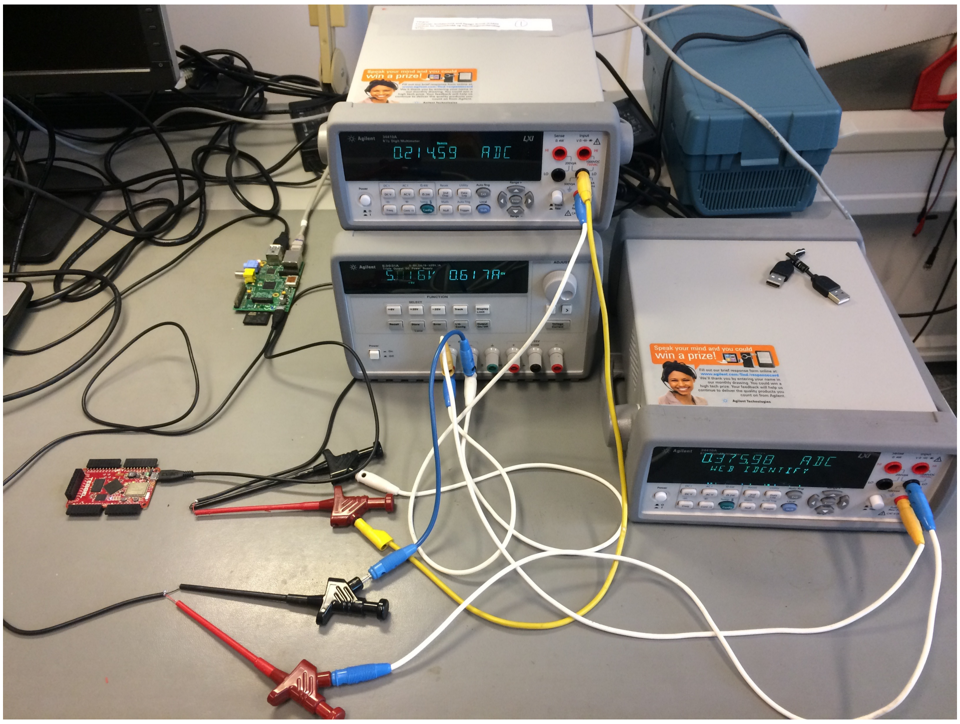


Figure 3.4: The hardware setup of the experiment

Unlike the Raspberry Pi, the Tessel hardware does not allow for remote access while it is running on external power because the USB port is the only communication port able to flash the Tessel. Instead, the program was written to the internal flash, which will start to run whenever the Tessel boots. Then the hardware was reset from software after each program run. This feature did not exist in the platform, but was written for this experiment, and has since been accepted into the project.[7] After the approximate wanted number of runs, the logging was stopped.

Data manipulation

The Benchvue Software can export the measurement data in CSV format, in two columns with a timestamp and the sampled data at that time. A Python program read these files and performed the required manipulations on the data.

[7] `https://github.com/tessel/t1-firmware/pull/140`

When the Raspberry Pi was sleeping, the current was always under 0.39 µA, and at all other times it was above. This threshold was exploited to create a stream with only the samples taken while a program was running. The stream was then split up into blocks of as many samples as the average program contains. To find the energy consumed during each program run, the blocks were added up. The average of these sums was then the average power usage of one experiment.

The current readings through the reset procedure were removed to get similar data from the Tessel. With the reset procedure not as clear as the sleep dips in the power readings of the Raspberry Pi, it was more difficult to remove the excess data. Instead of filtering the data below a threshold, the time data was used to remove the samples after each program has been run. By using the approximate running times in table 4.1 (page 27), and the timed reset of 12.8 s, it was possible to estimate the start and end of each program run. These start and end points were used to remove the samples in between, which corresponded to the reset procedure. Then the samples kept were used in the same manner as for the Raspberry Pi programs.

The average current drawn by each program was divided by 1,000,000 to show the current drawn by each iteration. The electric power used in each iteration was then calculated with the formula:

$$P = IV$$

This finds the amount of work done by the processor for each iteration in the benchmarks.

4 Results

First shown in section 4.1 are the base power usage of both tested hardware platforms, i.e. the current drawn when no program is running. The Raspberry Pi is shown to draw more than twice the current than the Tessel does.

Next, in section 4.2 the approximate time for running the various programs on each software platform are tabulated. These numbers were used to find the average current drawn by a program run.

Then in section 4.3, graphs of the current samples taken from the running of each program on every platform are presented, to illustrate the typical appearance of the current drawn during a program run.

In section 4.4, the average current drawn per sample is shown, allowing for direct comparison with the results from section 4.1. The results show that the Tessel draws far less current than the Raspberry Pi, but also that the Espruino VM draws less than io.js. The number of iterations per sample is shown in section 4.5, where it is shown that io.js manages to perform an order of magnitude more iterations per sample than the other platforms.

Lastly, in section 4.6, the calculated average current drawn and power used by a single iteration of each program on every platform is shown. Here it can be seen that despite drawing the most amount of current when running, io.js is the most efficient one per iteration.

4.1 Base Power Usage

In figures 4.1 and 4.2, power measurements of both hardware platforms running with no active program are graphed. All the noise that can be seen is various background programs running, like networking and resource management.

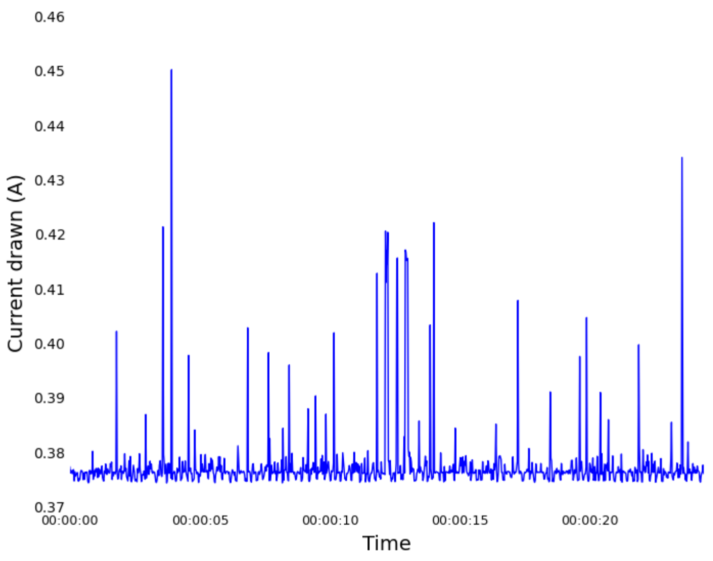


Figure 4.1: The Raspberry Pi running with no active program

With a sample set over 10 minutes, the average current drawn per sample is 0.378A on the Raspberry Pi with no running program.

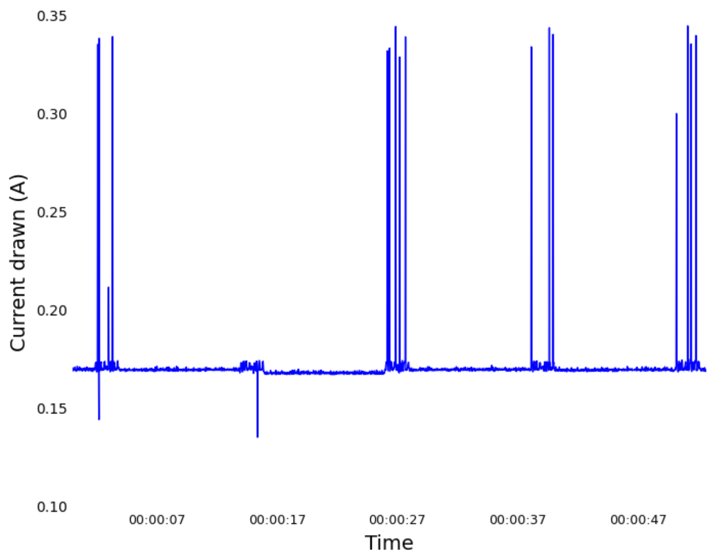


Figure 4.2: The Tessel running with no active program

In the Tessel graph, fig. 4.2, there is a lot less noise. The high readings that can be seen are from the WiFi-chip that the Tessel hardware provides. About every 12 seconds the Tessel gathers a list of all WiFi networks that are within range, and check if any of them is recognized.

With a sample set over 10 minutes, the average current drawn per sample is 0.162A on the Tessel with no running program.

4.2 Program time

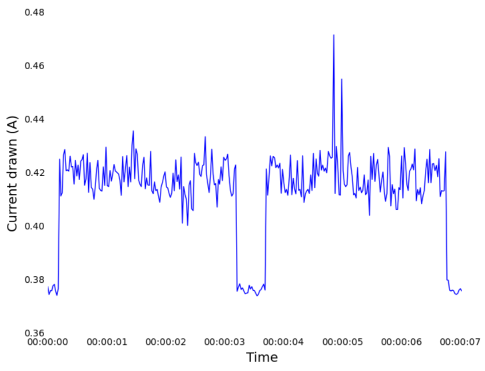
In table 4.1, the approximate times of running the programs are gathered. To note is the running time of the Left shift program on the Tessel, which is much higher than any other running time. The closure program also systematically uses a longer time on every platform.

	Addition	Multiplication	Left shift	Closure
Espruino	0m 48s	0m 48s	0m 49s	1m 19s
io.js	0m 3s	0m 3s	0m 3s	0m 4s
Tessel	0m 19s	0m 19s	6m 12s	2m 17s

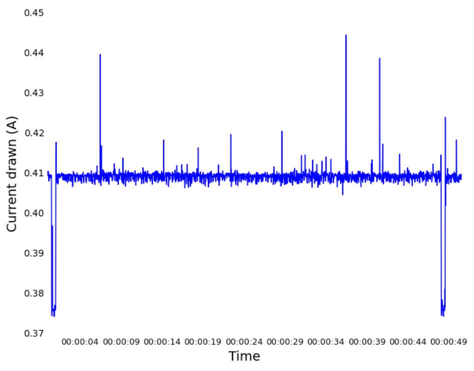
Table 4.1: Approximate time of each program

4.3 Current samples

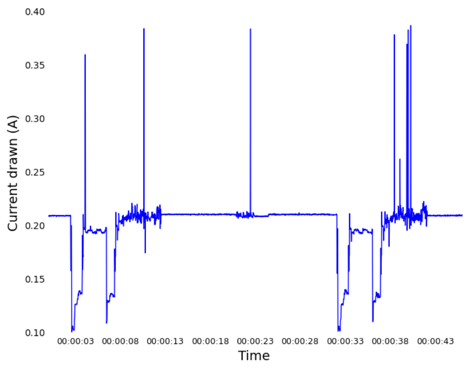
This section shows the results gathered by the multimeter, as described in section 3.4. Shown in figs. 4.3 to 4.6 are graphs of the samples of the four benchmarks running on each platform, cut to approximately one program run.



(a) io.js



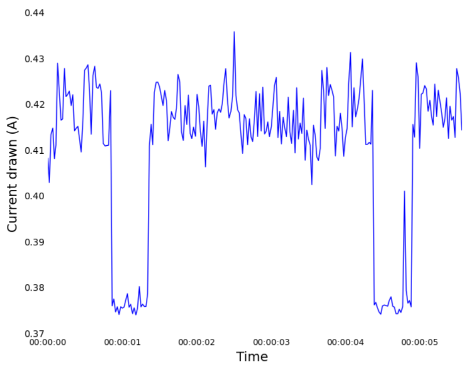
(b) Espruino



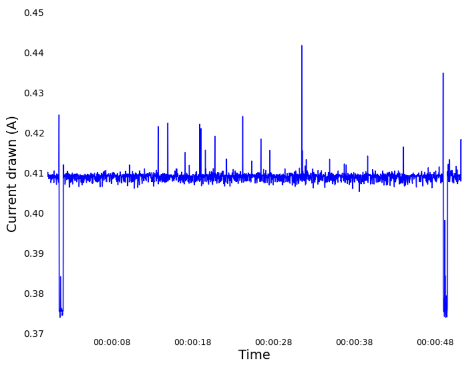
(c) Tessel

Figure 4.3: Samples of the Addition Program

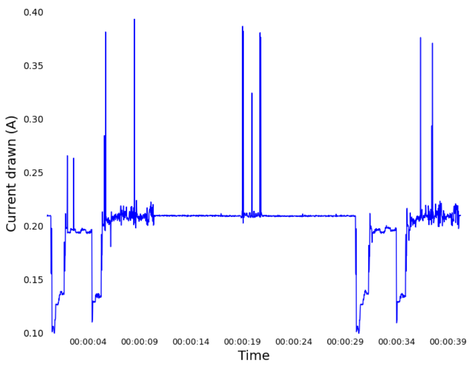
In fig. 4.3a, the Addition Program running on io.js is shown. The big dips that can be seen are the sleep command that is used to differ between each run. The same is visible in fig. 4.3b, where an Espruino run of the same program is shown. The spikes that can be seen in fig. 4.1 are not visible during the sleep procedure, due to other OS functions being paused.



(a) io.js



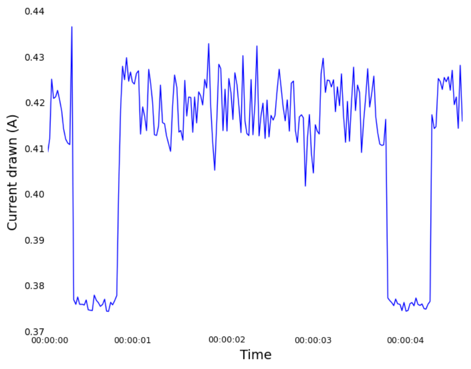
(b) Espruino



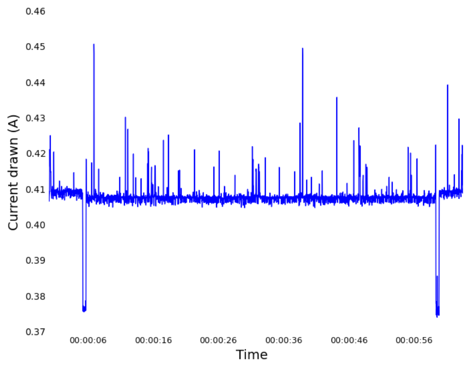
(c) Tessel

Figure 4.4: Samples of the Multiplication Program

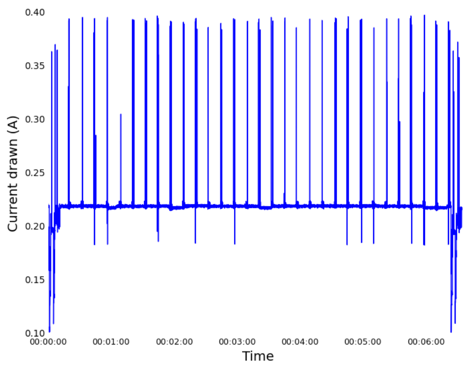
In figs. 4.3c, 4.4c, 4.5c, and 4.6c, the current samples taken of the Tessel is shown. In all of these figures, the current drawn during the reset procedure can be seen.



(a) io.js



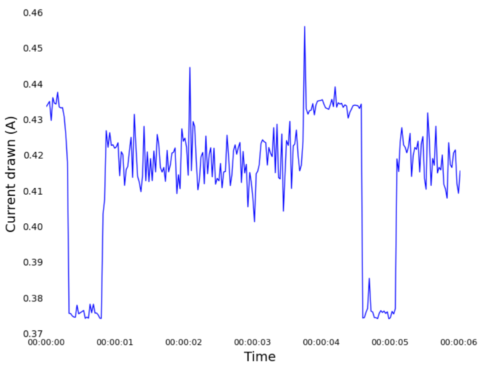
(b) Espruino



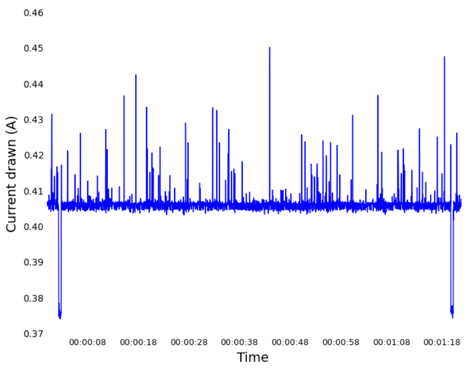
(c) Tessel

Figure 4.5: Samples of the Left Shift Program

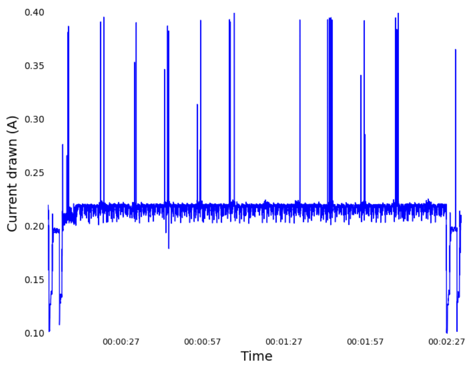
As the Left Shift program uses much longer time on the Tessel, fig. 4.5c is different from all others. The regularity of the WiFi chips activities is especially clear here.



(a) io.js



(b) Espruino



(c) Tessel

Figure 4.6: Samples of the Closure Program

Comparing fig. 4.6c with the other samples that were taken off the Tessel, the values vary more when running the Closure program. When translating the Closure program, more than one instruction must be used. From the figure, it is clear that some of these instructions uses less power than others.

4.4 Average current drawn per sample

Add		Multiplication	
Espruino	0.409 A	Espruino	0.409 A
io.js	0.419 A	io.js	0.419 A
Tessel	0.203 A	Tessel	0.203 A

Shift		Closure	
Espruino	0.409 A	Espruino	0.406 A
io.js	0.419 A	io.js	0.422 A
Tessel	0.219 A	Tessel	0.215 A

Table 4.2: Average current drawn per sample

Table 4.2 shows the average current drawn per sample of each program, with the same data as used in table 4.4. As can be seen, io.js draws the most current per sample.

4.5 Iterations per sample

The number of iterations done per sample is shown in table 4.3 using the number of samples taken in the experiments.

These values are tied to how fast the programs run, as a program that runs for a longer time will be sampled more often because the sample rate being constant.

Add		Multiplication	
Espruino	4.8	Espruino	4.7
io.js	64.0	io.js	64.1
Tessel	8.1	Tessel	7.8

Shift		Closure	
Espruino	4.6	Espruino	3.0
io.js	63.9	io.js	63.9
Tessel	0.7	Tessel	1.8

Table 4.3: Iterations per sample

4.6 Energy use per iteration

	Add			Multiplication		
	Tessel	io.js	espruino	Tessel	io.js	espruino
Current (μA)	158.18	55.286	844.32	165.67	55.276	844.16
Power (μW)	793.27	277.26	4,234.3	830.84	277.21	4,233.5

	Closure			Shift		
	Tessel	io.js	espruino	Tessel	io.js	espruino
Current (μA)	1,196.3	70.896	1347.9	3,334.5	55.276	862.05
Power (μW)	5,999.4	355.54	6,759.7	6,723	277.21	4,323.2

Table 4.4: Energy per iteration in loop

In table 4.4, the calculated average current drawn and power used per iteration in each program is collected. The values are shown in μA, thus as the number of iterations are 1,000,000, the values in the table also represent the average current drawn by the entire program run.

5 Discussion

In this chapter, the results shown in the last chapter are discussed, along with issues that arose with the method.

5.1 The benchmarks

The benchmarks in the experiment were chosen to test some JavaScript features, in a way that mimics the work done by Tiwari et al. But instead of just testing one instruction, JavaScript expressions are tested. The Closure program especially demonstrates this point, as a function definition and call uses many instructions in an assembly implementation. But also the other programs contains more than a simple operation, with the use of variables translating into memory accesses.

An optimizing compiler should be able to reduce the number of instructions in all of the benchmarks. For example by using constant folding, the variable lookup in each of the operations could be removed. After this constant folding, the expression itself could be found identical in every iteration of the loop, causing the compiler to move it out of the loop. The operation in the loop would then simply become a variable declaration. In the Closure program, the function could be inlined, resulting in the operation also being just a variable declaration.

5.2 Comparison between the platforms

This section compares the results of the different platforms, through the running times of the programs; the values found per sample and the calculated per iteration results.

Running time

Execution time on the Raspberry Pi is 3.89 times faster than on the Tessel, as the Raspberry Pi's clock runs that much faster. Adjusting table 4.1 on page 27 for this, the conjectured running times of the programs on the Tessel if the hardware was as powerful as the Raspberry Pi are shown in table 5.1. As can be seen, the Tessel speed of the Addition and Multiplication program is comparable with the io.js version. Listing 5.2 shows that the JavaScript operation is translated directly to the equivalent Lua operation, making the program run very efficiently. The cause of the massive performance loss in the Left shift program is described in section 5.4.

	Addition	Multiplication	Shift	Closure
Espruino	48 s	48 s	49 s	79 s
io.js	3 s	3 s	3 s	4 s
Tessel	4.88 s	4.88 s	95.6 s	35.21 s

Table 5.1: Adjusted running times

The Closure program uses a longer time on each platform, due to a function not being represented by a single instruction. As the V8 engine behind io.js includes JIT compilation, a lot of the extra work that a closure needs can be optimized.

While the values shown here can give an idea into the speed differences, the numbers are not an exact prediction. Other factors than just the clock speed might influence the running times on different platforms. For example the memory access speed, communication bus performance and peripherals that the hardware provides could have an impact.

Per sample values

As seen in table 4.2, io.js draws most current per sample, but just 0.01 A more in most cases. But by referencing table 4.3, it is clear why it is higher than the other two, but still uses less energy per iteration, as table 4.4 show. Because io.js executes the program much faster, a lot more iterations are done per sample.

Comparing these results to the average energy drawn by the platforms when running no program, section 4.1, the percentage of the current drawn by the program is shown in table 5.2. This table demonstrates that most of the energy consumed when running programs on the devices goes to powering the device. The operating system or additional hardware the platform provides, such as light emitting diodes (LEDs), can be examples that draw additional power.

Add	
Espruino	7.58%
io.js	9.88%
Tessel	20.2%

Multiplication	
Espruino	7.58%
io.js	9.88%
Tessel	20.2%

Shift	
Espruino	7.58%
io.js	9.88%
Tessel	26.0%

Closure	
Espruino	6.90%
io.js	10.4%
Tessel	24.7%

Table 5.2: Percentage of current drawn by program

When running programs on the Tessel, more of the current drawn goes to the program because the OS is specifically tailored to just running the programs on the Tessel hardware. The OS on the Raspberry Pi is a generic system created for everyday use on the device. It supports a graphical user interface and has many packages pre-installed. By changing to a more optimized OS, with fewer features enabled, the energy usage could be lowered.

Energy use

Comparing the energy use of the Tessel and io.js, shown in table 4.4, it is clear that the scenario illustrated in fig. 2.1 holds true. The energy used by running a program in io.js on a faster computer is lower than using Tessel on a slower one. That means if the need is to run the program using the least amount of energy, it is better to use io.js on the Raspberry Pi. But, as can be seen from section 4.1, if the faster device is running while the slower is finishing up, it will use more power overall. If the system cannot be powered down after it is done running, the energy saved while running the program will be for nothing.

As seen by Valluri and John (2001), optimizing for speed will lead to lower energy use. By looking at table 5.1 and table 4.4, there is a lot to improve on the speed of both the Espruino on the Raspberry Pi and the Tessel.

5.3 Why is Espruino so slow?

With the goal of the Espruino project being to create a low-power JavaScript embedded unit, it seems counter-intuitive that it uses so much energy when running on the Raspberry Pi. Seen from table 4.4, it is a power of magnitude worse than running io.js on the same platform. Why is this?

When the Espruino executes code, it does so directly from the source, evaluating one expression at a time. With the entire program being a loop, this leads to a lot of time being put into parsing the program. After 1,000,000 iterations, this parsing can add up to quite a lot.

Additionally, the memory model of Espruino does not scale with many lookups. As it is using a linked list to store variables, doing a lot of variable arithmetic, as all the programs do, leads to more lookups in this list.

Espruino cannot do optimizations, as it executes the code given to it. When developing code for the Espruino, one should take into account the limitations of the platform. It may not run all code fast, but code optimized for it should be better. A solution for runing generic code faster on Espruino could be to add some software that optimizes the JavaScript code before running it. While fixing some problems, this runs counter to the goal of Espruino to give the developer the ability to debug JavaScript that is running on the device.

5.4 The Shift program on Tessel

When running the Left shift program on the Tessel, it needs a lot more time than any other program and thus use a lot more power. The code running must be analyzed to find out why.

In listing 5.1, the entire code generated by the Colony Compiler, the JavaScript to Lua compiler used by the Tessel, of the Left shift program is shown. Comparing the code in the while loop with code generated for the Multiplication program, seen in listing 5.2, there is quite a significant difference between what the compiler produces for the different operations. Lua does not have an operator for the left shift operation like JavaScript does, but the same behavior is available through the bit-module. To maintain control over what is global variables in the Tessel firmware, bit has been renamed to _bit, which is why this is used when calling the shift operation. Therefore, the operation in line 8 of listing 5.1 is equivalent to the JavaScript operator.

```
return function (_ENV, _module)
local exports, module = _module.exports, _module;
local i, a, b, c = i, a, b, c;
--[[0]] i = (0);
while ((i)<((1000000))) do
--[[38]] a = (1);
--[[53]] b = (2);
--[[68]] c = _bit.lshift(_G.tointegervalue(a),_G.tointegervalue(b));
local _r = i; i = _r + 1;
end;
return _module.exports;
end
```

Listing 5.1: Lua code generated by the Colony Compiler for the Shift Program

```
--[[35]] a = (1);
--[[47]] b = (2);
--[[59]] c = ((a)*(b));
```

Listing 5.2: Lua code generated for the Multiplication program (excerpted)

As the ECMAScript standard requires the values in a shift expression to be cast to integers, the compiler explicitly casts the values to integers. (Ecma International, 2011, section 11.7.1) This casting is implemented in the framework, and can be seen in listing 5.3.

As seen, js_toprimitive is called on the value, to ensure that Lua only tries to convert a primitive to a number and thus avoiding error. The implementation of js_toprimitive can be seen in listing 5.4. While all the function calls are built-in Lua functions in the

```
_G.tointegervalue = function (val)
  val = tonumber(js_toprimitive(val))
  if val == nil then
    return 0/0
  else
    return math.floor(val)
  end
end
```

Listing 5.3: tointegervalue from the Tessel runtime code (Source: https://goo.gl/xwNXyZ#L43

```
local function js_toprimitive (val)
  if type(val) == 'table' then
    val = val:valueOf()
    if type(val) == 'table' then
      val = tostring(val)
    end
  end
  return val
end
```

Listing 5.4: js_toprimitive from the Tessel runtime code (Source: https://goo.gl/xwNXyZ#L23

end, adding extra instructions is hurting the performance. When performing a single bit shift instruction, two calls to tointegervalue are issued, and each of those calls js_toprimitive once, for a total of 4 function calls in the framework code. With the addition of the Lua built-in function calls, three per operand, as well as the two comparisons at least 14 instructions are added to ensure type safety for the left shift operation. This number assumes that only a single instruction is used per function call, which is a low estimate.

A way to remove a lot of this overhead when executing would be to let the compiler do optimizations to check if the operand is an integer at compile time, i.e. do constant folding. While this might hurt the compilation speed, the trade-off should be worth it to make the instruction faster, at least in some cases.

But there is no need for the compiler to do any type validation here. The implementation of JavaScript in the Colony compiler tries to follow the JavaScript specification, and the shift implementation is exactly after the standard. Lua, just as JavaScript, is a dynamic and weakly typed language. The left shift operation in Lua works on exactly the same numbers as the JavaScript operation. In the current implementation, the casting to integers is done twice, adding at least twice the amount of work needed. If the compiler had taken into account how the Lua interpreter executes the operations, much more energy could be saved. With the finding of this expression with a bad implementation, it is not hard to imagine that there are other operations in the framework that also will waste energy. By optimizing the compiler for creating better Lua code could reduce the power usage.

5.5 What does current draw mean?

Why is measuring the current drawn interesting when investigating energy use of a computer? The power is a measurement of the amount of work done per second. When the computer gets more instructions to do, i.e. a program is started, the amount of work done increases. As the computer cannot generate energy from nothing, no known mechanism in the universe can, it needs to get the energy from other sources. The electric power formula,

$$P = IV$$

shows that to increase the power, either the current or the voltage have to increase. Since the power source keeps the voltage constant, the only way for the device to increase the power is to draw more current. The increase seen in current drawn when starting a program is corresponding to the increased work done by the computer.

5.6 Documentation problems

ptxdist on Giant Gecko

As mentioned in chapter 3, the experiment was planned to be done on the Giant Gecko as well, by using PTXdist. While setting up the distribution and installing it on the development board was easy enough, getting the software to run the experiments was much harder.

Tried first was downloading a pre-compiled version of io.js for ARMv7 architecture, which the official io.js website provides. This version was copied to the binary folder of the operating system on the Giant Gecko and given running permissions. When io.js then was started through the shell access, an error occurred, saying “applet not found.” It seemed that *Busybox* caused this error, which is a program that provides normal shell programs to minimal systems. Busybox tried to execute the command as it was a program supplied by Busybox, but it could not find the program since it was not a Busybox applet.

One possibility was that there was some issue with the binary file. If so, the solution would be to cross-compile the program. When cross-compiling the program, an issue with library files not being present arose. The explicit locations of the libraries were used to combat this when building, which fixed this issue. But after copying the compiled file to the device, the Busybox error was persistent.

Neither the documentation nor any web searches yielded answers to what the problem was. When asking people associated with PTXdist, they could not answer to why this happened. A guess to what happens is that Busybox somehow is invoked first when running other programs. To actually run programs that are not a part of Busybox, some setting that tells Busybox to not run its applet must be set, but if this is the case, it is not documented anywhere.

Instead of wasting more time trying to get programs to run, it was decided to focus on the platforms that were already running. This anecdote would be an example of how hard it is to use microcontrollers.

Tessel runtime on Raspberry Pi

Another planned test to run was to use the Tessel runtime on the Raspberry Pi, allowing for better comparison between the hardware platforms. But the Raspbian Linux distribution has not been updated in a long time, due to the Raspberry Pi 1 having an ARMv6 processor instead of ARMv7 that Debian supports natively. A lot of the dependencies needed to build the Tessel runtime were out of date, making the build process hard to complete. Not being the primary target for the Runtime, and as the Raspberry Pi is a slow development platform, there exists no documentation on how to fix the dependency issues.

The Raspberry Pi 2, just released this spring, has an ARMv7 processor, which the standard Debian distribution supports.(Tessel 2 Hardware Overview) With the new hardware, up to date packages built for the Debian ARM distribution will become available. After updating the dependencies, the build process should be no problem.

5.7 Error in logged measurements

When analyzing the data from the Keysight Benchvue application, a bug showed itself. For some data sets, when manipulating the timestamps of the measurements, many of the samples were not included. When graphing this data, an arbitrary cut emerged. Looking at the data, it was clear that the data formats were in 12-hour format, where the hour field went from 01 to 12, and then back to 01. When some of the experiments went on through either noon or midnight, the continuity of the samples was broken.

The affected lines in the result files were changed using a simple search and replace tool. As only the hour field needed replacement, and the maximum theoretical strings that needed to be changed was 12, this was not automated, but simply carried out for each data set required. There should be no issue with the results because of these edits.

There might be some settings in the application to use a 24-hour clock when writing the time stamps. But as the solution described above was satisfactory, it was decided to not throw away the data already acquired.

6 Conclusion

In the following chapter, the conclusions that this thesis can draw are mentioned. Broadly, measures that decrease the number of instructions in a program will lead to lower power usage on embedded platforms. Finally, some ideas for future work are presented.

By the discussion in section 5.4, it can be concluded that the Colony Compiler which translates JavaScript to Lua in the Tessel framework can save energy. Firstly, by outputting Lua code that does not have to do more work than necessary, and secondly, by utilizing common compiler optimizations.

An optimizer for the Espruino VM, outputting JavaScript code optimized for the special needs of the framework, could lower the energy use on the platform. This is only necessary if there is a need to run standard JavaScript programs. When developing new programs for the platform, these limitations should be taken into account, as this can save energy.

If the goal is to use less energy on a platform, an area to look at is to minimize the power used by the operating system. Table 5.2 show that most of the power used is by the OS, even on the Tessel, which has a device-specific OS.

While both hardware platforms tested in this thesis are embeddable, they are quite different. The Raspberry Pi is a more powerful device, and can execute the programs faster, but at the cost of higher power usage. As the total energy used during the execution of a program was lower on the Raspberry Pi with io.js, using io.js might actually save energy. However, if the device continues to run after the program has finished, the costs outweigh the initial gains. Even if it is possible to turn off the system after execution is over, the cost of turning the device on again might also be higher than what was saved during execution. Thus, the choice of device when designing for saving energy needs to be a balanced decision.

The Tessel 2, released in the fall of 2015, will use io.js. A conclusion from this thesis is that it might not increase the power consumption of the new device, compared to the Tessel 1.

The JavaScript engine that uses the least amount of power per iteration in the experiment is the fastest one. This suggests that optimizing for speed is also optimizing for lower energy use. Together with a basis in the literature, the recommendation of this thesis is to focus the design of embedded frameworks to execute faster in order to lower the power consumption.

The results of the projects suggests that the answer to the research question, “what are energy efficient ways of bringing high-level programming languages to embedded computer systems?”, is that focusing on creating a framework that executes programs as fast as possible is the most energy efficient.

6.1 Future work

According to the research in this paper, more tests are warranted. As the energy use of the different expressions tested varied as much as it did, it would be interesting to test more operations on the different platforms. With the Tessel implementation shown to be not optimal, further tests of other expressions could find other parts of the framework that can be improved.

To confirm the suggestion that the engines tested in this project would use less energy if the JavaScript code was optimized, more tests could be done. For example, perform an identical project as laid out in this report, only using the benchmarks optimized as in section 5.1. Besides using benchmarks that has been manually optimized, an optimizer for JavaScript could be developed, allowing for tests of automated optimizations.

A way of automating the experiments on the Tessel, would be to connect it to a computer using another modified USB cable. A USB cable carries its signal through four cables, two are for Vcc and ground, while the last two are for the data connection. There should be no problem in powering the device from an external power supply, as done in the experiment, and at the same time delivering data from a computer.

The benefits of this are to get the same control over the programs run as through the shell of the Raspberry Pi. This would remove the need for resetting the Tessel from the software itself, making it easier to see where the program runs in the sample data.

However, there is no guarantee that this will work, as it is undocumented in the specification. There should be no connection between the power and data lines when using a USB cable, but implementations might vary. To use this technique, testing needs to be done. Because of timing limitations in this project, the solution described in chapter 3 was decided to be sufficient.

With both the Raspberry Pi 2 and the Tessel 2 being released in 2015, the experiment done in this project could be tested on those platforms as well. Also, testing the Espruino Pico hardware, in the same way, could yield interesting results as this would be comparing with a truly low power device.

Bibliography

V8 Design Elements. `https://developers.google.com/v8/design`. Accessed: June 23, 2015.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, & Tools*. Pearson Education, 2 edition.

Aycock, J. (2003). A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113.

Csere, C. (2012). 2013 Tesla Model S Test. `http://www.caranddriver.com/reviews/2013-tesla-model-s-test-review`. Accessed: June 14, 2015.

De Lima, E., De Souza Xavier, T., Faustino da Silva, A., and Beatryz Ruiz, L. (2013). Compiling for performance and power efficiency. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2013 23rd International Workshop on*, pages 142–149.

Ecma International (2011). *ECMAScript Language Specification*. `http://www.ecma-international.org/ecma-262/5.1/Ecma-262.pdf`.

Fortuna, E., Anderson, O., Ceze, L., and Eggers, S. (2010). A limit study of javascript parallelism. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10.

Grimm, C., Neumann, P., and Mahlknecht, S. (2013). *Embedded Systems for Smart Appliances and Energy Management*. Addison Wesley.

Jahre, M. (2013). SHMAC: An Infrastructure for Heterogeneous Computing Systems Research. `http://www.ntnu.edu/documents/139931/80945963/eecs-shmac-hipeac-csw.pdf/d6c2b771-0c40-47d4-a3e9-17b5e9f0d58d`. HiPEAC Computing Systems Week, Tallinn.

Joyent. About Node.js. `https://nodejs.org/about/`. Accessed: June 23, 2015.

Kavvadias, N., Neofotistos, P., Nikolaidis, S., Kosmatopoulos, C., and Laopoulos, T. (2004). Measurements analysis of the software-related power consumption in microprocessors. *Instrumentation and Measurement, IEEE Transactions on*, 53(4):1106–1112.

Kolker, E. Tessel 2 hardware overview. `https://tessel.io/blog/113259439202/tessel-2-hardware-overview`. Accessed: June 22, 2015.

Kopetz, H. (2011). Internet of things. In *Real-Time Systems*, Real-Time Systems Series, pages 307–323. Springer US.

Kumar, R., Farkas, K., Jouppi, N., Ranganathan, P., and Tullsen, D. (2003). Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81–92.

Le Sueur, E. and Heiser, G. (2010). Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, pages 1–8, Berkeley, CA, USA. USENIX Association.

lua.org. About Lua. `http://www.lua.org/about.html`.

McKay, J. Moving faster with io.js. `https://tessel.io/blog/112888410737/moving-faster-with-io-js`. Accesed: May 7, 2015.

Ortiz, D. and Santiago, N. (2008). Impact of source code optimizations on power consumption of embedded systems. In *Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on*, pages 133–136.

Pengutronix (2014). How To Become a PTXdist Guru. `http://www.pengutronix.de/software/ptxdist/appnotes/OSELAS.BSP-Pengutronix-Generic-arm-Quickstart.pdf`.

Raspberry Pi Foundation. Power Supply. `https://www.raspberrypi.org/documentation/hardware/raspberrypi/power/README.md`.

Raspberry Pi Foundation. Raspberry Pi Frequently Asked Questions. `https://www.raspberrypi.org/help/faqs/`.

Russell, J. and Jacome, M. (1998). Software power estimation and optimization for high performance, 32-bit embedded processors. In *Computer Design: VLSI in Computers and Processors, 1998. ICCD '98. Proceedings. International Conference on*, pages 328–333.

Ryan, T. A new engine for your tessel. `https://tessel.io/blog/102381339917/a-new-engine-for-your-tessel`. Accessed: May 7, 2015.

Salomon, D. (1992). *Assemblers and Loaders*. Ellis Horwood, Upper Saddle River, NJ, USA.

Silicon Labs (2014a). *EFM32GG Reference Manual*. `http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32GG-RM.pdf`.

Silicon Labs (2014b). *EFM32GG990 Datasheet*. `http://www.silabs.com/Support%20Documents/TechnicalDocs/EFM32GG990.pdf`.

Taylor, M. B. (2012). Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1131–1136, New York, NY, USA. ACM.

Technical Machine. Powering Tessel. `https://tessel.io/docs/power`.

Technical Machine. Tessel hardware documentation. `https://tessel.io/docs/hardware`. Accessed: June 23 2015.

Tiwari, V., Malik, S., and Wolfe, A. (1994). Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445.

USB Implementers Forum, Inc. (2007). *Universal Serial Bus Micro-USB Cables and Connectors Specification.* `http://mgvs.org/public/shema/datasheet/usb_20/Micro-USB_final/Micro-USB_1_01.pdf`.

Valluri, M. and John, L. K. (2001). Is compiling for performance — compiling for power? In Lee, G. and Yew, P.-C., editors, *Interaction between Compilers and Computer Architectures*, volume 613 of *The Springer International Series in Engineering and Computer Science*, pages 101–115. Springer US.

w3. A short history of javascript. `https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript`. Accessed: May 12, 2015.

W3Techs. Historical yearly trends in the usage of client-side programming languages for websites. `http://w3techs.com/technologies/history_overview/client_side_language/all/y`. Accessed: May 12, 2015.

W3Techs. Historical yearly trends in the usage of client-side programming languages for websites. `http://w3techs.com/technologies/history_overview/programming_language/ms/y`. Accessed: May 12, 2015.

Weissel, A. and Bellosa, F. (2002). Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 238–246, New York, NY, USA. ACM.

Williams, G. (2015a). Espruino Interpreter Internals. `http://www.espruino.com/Internals`. Accessed: June 23, 2015.

Williams, G. (2015b). Espruino Performance Notes. `http://www.espruino.com/Performance`. Accessed: June 11, 2015.

Wolf, W. (2008). *Computers as Components: Principles of Embedded Computing System Design.* 2 edition.