A hybrid method of static and dynamic energy analysis of external hardware components on LLVM IR

Mark Westenberg

Open University, Valkenburgerweg 177, 6419AT Heerlen, The Netherlands
info@markwestenberg.nl

Abstract. Systems that consume resources are often controlled by software. Inefficient code and bugs in software can have a great effect on resource consumption. Static analysis of software does not require execution and does not need a complicated measurement setup. For this reason it is easier to integrate static analysis into the (software) development life cycle than dynamic analysis. To allow programmers to test the sustainability of their software in an incremental way we developed a hybrid method that combines dynamic with static analysis. By creating a program trace on the lower level language LLVM IR we can support multiple higher-level languages and achieve higher precision in terms of timing. This method is validated in a test setup where actual measurements are made and compared to the calculated consumption.

Keywords: energy analysis · static analysis · dynamic analysis · LLVM · hardware components.

1 Introduction

We use all kinds of electronic devices each day that consume energy. Consumers and businesses alike are aware that hardware should be energy efficient. In the European Union an energy labeling system was introduced that allows consumers to compare the efficiency of hardware like washing machines, dish washers, heating systems and even cars. A higher rating label justifies a higher price and allows for more profit for the company selling the hardware, it also has a cost benefit for the consumer.

It is often forgotten that hardware is controlled by software and it is the software that influences the energy efficiency of hardware. In case of the Volkswagen emissions scandal, software was used to detect if a car was being tested. Software would then put the engine in a lower emission compliant mode to reduce the NO_x emissions when measured. Although, this was done on purpose and with ill intend, inefficient code and bugs in software can also lead to efficiency loss

in terms of energy consumption. It is thus useful to write software that makes efficient use of resources from an environmental and economic perspective.

Building and testing software nowadays, is often done in an agile way. This means software is built in small increments where building and testing are a continuous cycle in relatively short periods of time. When programmers want to test the sustainability and efficiency of their algorithms this must be done quickly and should be easy to setup and fully automated. Dynamic analysis is hard to automate because execution is required making it more difficult to compare results with previous runs automatically. In case of high energy consuming devices or machinery coming up with efficient ways of testing is a daunting and costly task. Static analysis is much more efficient and easier to automate but less precise than dynamic analysis.

In developing our approach we wanted to come up with an easy way of providing feedback to programmers on the energy or resource efficiency of their algorithms. Our focus was not on the energy consumed by the software itself but rather on the external hardware components the software controls. Now (because of this paper), it is possible to integrate resource consumption analysis easily in the existing software development life cycle that is fast to setup and low in cost. For example, when software is made ready for deployment automated unit test cases are executed and must all pass before an approval is given for production usage. As part of this process it would be desirable to add a fully automated test for energy efficiency. When changes to software are committed to the repository and tested before deployment an automated resource consumption measurement should be taken and compared to certain thresholds or even previous test runs.

In this paper we present a new approach by combining dynamic and static analysis to calculate the energy consumption of external hardware components. This research is based on earlier research in which an analysis is performed on an illustration language called ECA [6, 9]. Using our method it is possible to integrate energy analysis of hardware models into the development life cycle without the need for a complicated measurement setup. The method is composed of three identifiable steps. The first step is to create abstractions of the hardware components that must be implemented into code. This is done via function declaration annotated with time-dependent, time-independent or incidental energy consumption and the time (in milliseconds) that is required to call the external function that controls the external hardware component. A crucial part in this first step is discretization of the domain of possible input values that defines the trace of the program to make it not only deterministic but also terminate upon execution (see section 2).

We then compile the code to LLVM IR, a target independent intermediate representation that has a close resemblance with the assembly language [12]. To compile higher level code into LLVM IR a frontend compiler is required. Currently there are frontend compilers available for wide variety of languages including C, C++, Objective C, Swift, Ada, Haskell and even Java bytcode. LLVM IR

is in Single Static Assignment Form which is required to increase precision in terms of timing when calculating time dependent consumption (see section 2). The LLVM IR code is transformed by a tool and compiled into bitcode to create a trace of basic blocks and to remove all abstractions that cannot be executed. This transformed program is then executed and creates an ordered trace of basic blocks (see section 3).

The final step is to statically analyse the program calculating the energy consumption of the external hardware components using the abstraction from step 1 and the traces in step 2. In section 4 we describe the analysis method in more detail including the calculation approach of time-dependent and time-independent consumption. For this calculation we rely on previous research where a similar calculation is done on a language called ECA [6] (see section 7).

To test the validity of our approach a test setup was created with two Arduino's, a relay that can handle 230 Volt AC and a 20 Ampere current meter. We developed custom made software to measure the energy consumption in number of Joules to increase the accuracy of our measurements (see section 5). We then measured the energy consumption of a 75 Watt light bulb and 550 Watt vacuum cleaner controlled by software running from an Arduino. The measured results are compared with the calculated consumption and proved to be accurate in most cases (see section 6).

2 Hardware abstractions

Measuring energy consumption of hardware components can be done in several ways. The first and most direct approach is dynamic analysis. In case of high energy consuming components where small changes in code can have a great effect on the total energy consumption, it would be preferable to take measurements after each incremental change made in the control software. This makes dynamic analysis cumbersome and often comes at a high cost. For instance, the component might be simply too large, use enormous amounts of energy to operate or are simply not available for testing.

In our method we rely on static energy analysis as described in [6]. This method describes a way to calculate energy consumption by analysing a language called ECA. In this method hardware components consist of a component function and a component state. When the state of a component is changed in code the power consumption changes as well. Hardware components can have time-dependent and time-independent energy consumption and a time that is required to complete the function. This particular distinction is used in our method as well.

When statically analysing code to calculate energy consumption, timing is an important factor. The ECA language is a higher level language and specifically

designed to calculate energy consumption and not to be used in real world situations. Analysing languages like C or C++ is difficult because they have more complex grammar than ECA and are still very different from the actual operations executed by the CPU. To increase timing energy analysis is often done on a lower level [4, 7, 11] like an assembly language. The problem with assembly is its architectural dependency which makes it difficult to create a more generic approach. With LLVM IR we are not at assembly level but are getting very close to it and have the advantage of language and target interdependency. Of course, we still loose some of the precision when the code is further compiled but timing is still greatly increased compared to higher level languages like C.

When calculating the time of a single instruction it must first be transformed into SSA. A simple example of this is shown in listing 1. When this instruction is transformed to SSA (see listing 2) the number of instructions is tripled. However when compiled and optimized to LLVM IR only two instructions remain (3). These kind of optimizations are done by the front-end compiler which in our case was Clang with optimization level 3 (O3).

```
// a = 3
int x = a*a + a*a; // result: 18
```

Listing 1: *C-code*

```
a1 = a * a;  // 9
a2 = a * a; // 9
x = a1 + a2; // 18
```

Listing 2: *SSA Form*

```
%b = shl i32 %a, 1 // 3*2 = 6
%c = mul i32 %b, %a // 6*3 = 18
```

Listing 3: *LLVM IR*

```
imul eax, eax // 3 * 3 = 9
add  eax, eax // 9+9 = 18
```

Listing 4: *Assembly*

The assembly code in listing 4 is using a different approach then the LLVM IR code does. In the example we are lucky because a *bitshift left* requires more or less 1 cpu cycle which is similar to an *add* instruction. One can imagine situations where this similarity is less than in this example. That these differences might cause a noise in timing can be seen in the results in section 5. This example does show that lowering to LLVM IR increases precision and removes the necessity to convert all operations into SSA form because this is done for us by the front-end compiler.

To measure external hardware components several properties are required to calculate resource consumption. Although in this paper the focus lies on energy consumption other sources of resource consumption (i.e. water or gasses) can be measured as well. The only required properties are three numbers that indicate a time-dependent consumption, a time-independent consumption and a time unit that expresses the duration of the component state change.

These resource consumption properties are annotated in our code and linked to the external hardware components as declared but undefined functions. We used Clang to compile our code to LLVM IR but ran into the problem that functions that are not defined are removed by the compiler and with it its annotations. Defining the function with dummy data would cause the compiler to optimize the code, add it inline and change the timing. To prevent this we declare the function with annotation and define it with an underscore as prefix with a dummy implementation. In the code we call the undefined function but add the annotations to the defined function. This is corrected eventually by the energy analysis tool that performs the calculation.

Because this process is cumbersome we created a header file with macro definitions to set this up easily. A component must have a name and a declaration for each component state. In listing 5 an example is shown of a light bulb that can be turned on and off. Because we need to distinguish between two component states we add two declarations, one for each state. The light bulb uses 75 Watts

```
1  //Annotations using Macro's
2  ENERGY_ARG(digitalWriteHIGH, void, "LightBulb,75,0,0.004534",int,int);
3  ENERGY_ARG(digitalWriteLOW, void, "LightBulb,0,0,0.004534",int,int);
4  ENERGY(delay, void, "delay,0,0,5000");
5
6  //example implementation
7  const int relayPin = 7;
8  digitalWriteHIGH(relayPin, 1);
9  delay();
10 digitalWriteLOW(relayPin, 0);
```

Listing 5: *Implementation of components with energy annotation*

(time-dependent consumption), has no time-inde-pendent consumption and the function that triggers it is called digitalWrite and uses 0.004543 milliseconds to complete. The name of the component is given to keep track of any state changes. In this case we go from 75 to 0 Watts. The delay function of 5000 milliseconds is an example how to add a time delay. We do not want to execute the delay during the partial execution process but we do need the actual delay for the calculation.

The next step is decide how long and when the external component is changing state. Whether or not this happens depends on the component and its environment. Because we want to execute the program it has to terminate thus a finite set of input parameters must be given. One important thing to note here is that when input parameters are declared inline the front-end compiler will optimise the code to a larger extend then is often desirable. To prevent this the finite set should be given as input parameter and might cause some change in the program. When these changes our all part of the setup before the first energy

drawing consumption occurs this is not a problem and will not effect the final outcome.

To create a finite set of an otherwise infinite set requires discretisation of input values that influence the component state. In some cases the number of component states is almost infinite. Imagine a screen of a smart phone that can adapt the brightness of a screen between 0 and 100 where at 100 it is at full brightness and at 0 it is off. A sensor measures the surrounding light in the environment and changes the brightness accordingly. Phones are not always used so the usage of a phone depends on its user. We can analyse phone behaviour of a group of people and gather this data and put it in a single chart so we end up with scattered data points between 0 and 100. We can use discretization to create a finite set of values that will make up our test set. In listing 6 we show how with a few lines of code how this can be achieved. The amount of steps in this function must

```
int stepsize = data.size() / steps
std::sort(data.begin(), data.end(),compare);
for (int i = 0; i < steps; i++)
    threshold[i] = data.at(((i+1) * stepsize)-1);
```

Listing 6: *Discretizing input values in C++*

be carefully chosen. In case of a smartphone the stepsize granularity is directly linked to a time unit. Using a step-size of hours instead of 15 minutes can have a great impact on the final result. In most cases one can say that the higher the granularity the higher the precision but also the longer it takes to calculate the energy consumption. The reason for this is because the trace has to be created and the longer the trace the longer the path traversal through the program will take.

When the program is finished and all annotations are in place the code can be compiled to LLVM IR. To do this a front-end compiler is required. There are many front-end compilers available for a wide variety of languages. In our setup we used Clang to compile C-code because it is the language that is being used on an Arduino but any other language will work provided there is a front-end compiler available. When compiling C-code to LLVM IR there are several levels of optimisations that can be used. Because we wanted to increase precision as much as possible we chose for the highest level (O3) currently available.

3 Partial Execution

When statically analysing code to calculate energy consumption, timing is an important factor. In a case where loops and free variables are introduced this becomes a challenge. In figure 7 a simple example is given where a light bulb is

turned on or off based on the branch that was taken. In node A the light will be on and in node B it will be turned off. A condition at the end of node S decides

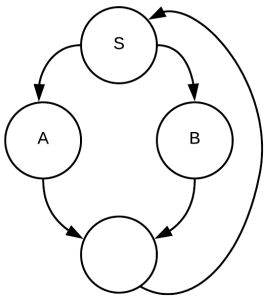

Fig. 7: *Loop example where each circle depicts a basic block containing multiple instructions. Node A will set the component state to* on *and B to* off.

which path is taken. The number of iterations is defined by a free variable n and unknown during static analysis. Incrementation or decrementation of n can take place in any of the nodes.

This example shows it is difficult to predict the component state and with it time-dependent power consumption. If the condition would be $n < 10$ and we know that $n \in \mathbb{Z}$ we must still evaluate each individual instruction and follow all possible paths to calculate the time-dependent energy consumption of the light bulb. The final result will be a formula that defines different scenarios for our energy consumption under certain conditions that have a dependency on variable n.

Another problem that must be solved when using static analysis is the evaluation of satisfiability of branches as is done in the study of the satisfiability modulo theories (SMT) problem. This can be explained by taking a simple branching example (see listing 8). If the type of the variables x and y are integers there are

```
1 (x * y == 2) ? A() : B();
```

Listing 8: *Branching condition example*

only 2 possible conditions that would make the condition true: $-2 * -1$ and $2 * 1$. When the type is changed into a rational number (float or double), evaluating the satisfiability of the condition is more complex. This is because the values of x and y can have an almost infinite amount of values that makes the condition true (i.e. $0.25 * 8$, or $0.5 * 4$ etc.). To know for sure that the branch is taken, or to find out under which conditions the branch is true or false the SMT solver must calculate all possible conditions under which this condition can become true (or false).

In the hybrid method we propose a concrete method to calculate energy consumption eliminating all issues that have to be dealt with when using an SMT solver. To do this we transform the program and partially execute it to create a trace that can be used for further analysis. This trace is created based upon input parameters as explained in the previous section that in turn decide not only the number of loop iterations but also which basic block is executed and in what order.

To execute the code and create a trace the code must be transformed to make it executable. All functions that are externally defined and declared in the translation unit must be removed before execution. This includes the with energy properties annotated component funtions. Also all uses of the registers that depend on the results of these functions need to be replaced. A trace generator tool was written on top of the LLVM Framework that removes all external functions calls and adds a print statement at the end of each basic block right before the the branch instruction. Because this transformation is done on the compiled code (LLVM IR) the print statements will stay in the exact location as inserted. The code is then compiled into LLVM bitcode and interpreted by *lli* an LLVM interpreter tool that is part of the LLVM tool chain. When executing the bitcode the discretised input is used as program argument (see section 2). The output is printed to a file that functions as program argument for the static energy analysis tool (see section 4).

4 Static Energy Analysis

For the final step an analysis tool was written that requires the orginal LLVM IR code (the one created before the transformation as described in the previous section), the program trace and the clock speed (in MHz) of the CPU the program will be running on. The tool creates an ordered vector of the program trace that can be matched to the LLVM IR code. Then it will traverse the code until it locates the first hardware component state change that consumes energy. From that moment each step of the way the analyser must keep track of the execution time of each instruction and other components that draw power. We assume that an external component C can have time-dependent C_{td} and time-independent (or incidental) C_{ic} energy consumption in Joules plus an execution time C_t that defines the time in microseconds that is required for the component to change state. With the trace as input we can calculate the energy consumption of each component and the sum of all components.

The trace that was created as described in the previous section is used to traverse the LLVM IR code until an annotated component function CF is encountered. From that moment for each instruction I the cost in cycles I_c is retrieved from the cost model. A power draw of a component is terminated when a time-dependent energy consumption of 0 is registered. The clock speed of the processor S in MHz is used to calculate the time in microseconds (µs) for each individual instruction

that is executed after a power draw. This leads to $I_t = I_c/S$ and depicts the costs of an individual instruction in µs. The component that starts a power draw has three attributes. These are the amount of power drawn in Watts (time-dependent) denoted as CF_{td}, a time-independent or incidental power draw C_{ic} and a time CF_t to execute the component function that changes the component state.

Expression 1 depicts the total energy consumption of a single component where n is the set of instructions or components that influence the time and i is a single occurrence. For instance a light bulb can be turned on and then another component state change can occur before it is turned off. The sum of all component changes CF_t that occur during a power draw must be added to the multiplication. This expression is basically applied to the entire vector that keeps track of active component functions.

$$CF_{td}CF_t(\sum_{i=1}^{n} I_{t_i} + CF_{t_i}) + CF_{ic} \tag{1}$$

Loops can slow down the calculation process significantly. For this reason when a loop is encountered the previous iteration can be added to the total sum instead of calculating the cost of the entire iteration. This is possible because we know what the next path through the program will be.

To calculate the total energy consumption that is produced by a program, state changes of each component and the time that passed in between those changes must be tracked. Each time an instruction is encountered the set of power consuming components is checked and the time is multiplied by the power draw (time-dependent consumption) and stored for each individual component. Apart from the total power in Joules per component a total power consumption of the entire program is also stored and displayed when the analysis is complete.

The final output of the analysis will be the time in seconds for each component and state, the total number of Joules consumed and a calculated wattage for each component.

5 Measurement setup

To test the validity of our method a test setup was created. Because the focus is on external hardware components we chose an ATmega328p microcontroller that is plugged into an Arduino and runs at 16 Mhz. The Arduino comes with an easy to use IDE and can be programmed using C as programming language. Before we could start testing we had to find a way to measure the energy consumption. There are many commercial products available but they only measure in KWh which was not fine grained enough for our purposes since measurements had to be done in Joules. To create a Joules meter a ACS712 20A current sensor was

used connected to a 230 Volts AC socket and a separate Arduino that would run the measuring software. Also a LED display was attached to display the voltage and amperage. For the external hardware component a vacuum cleaner of 550 Watt and a light bulb of 75 Watt was used that could be controlled by a relay (see figure 9). These two components were chosen for their availability and how they consume energy. The light bulb has a more constant power draw whereas the motor has more fluctuations in its power consumption.

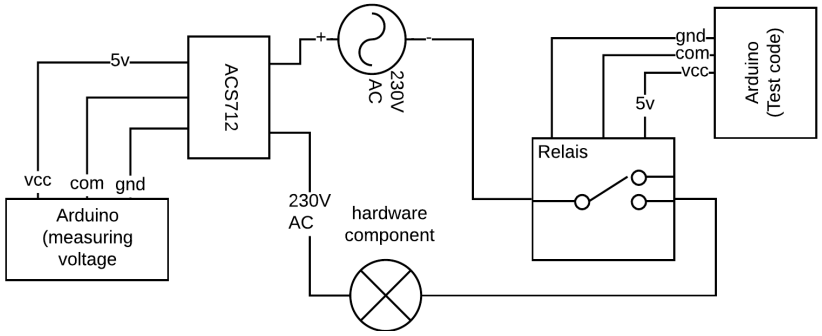

Fig. 9: *A schematic of the used test setup using two Arduino's and an ACS712 20A current sensor*

Alternating current (AC) is an electrical current that frequently reverses direction in a sinusoidal waveform. The voltage V we measure is a root-mean-squared (RMS) value an denoted as V_{rms}. The RMS is basically the equivalent value of DC power in terms of how much heat it can produce. In other words, when we speak of 230 V_{rms} this means it is the equivalent of $230V$ DC power. There are two ways to calculate the V_{rms}, the so-called the mid-ordinate rule and the analytical rule. For this paper it is too detailed to go into both methods but it suffices to say that the mid-ordinate rule uses samples instead of a constant value to determine the peak voltage value. We tried both methods and the mid-ordinate rule produced more accurate results and was therefore the obvious choice for our case study.

When measuring the voltage from the ACS712 we got a number between 0 and 1024 where the x-axis of our sinewave is located at 512. To retrieve the the mid-ordinate we need to deduct 512 from each sample and multiply it by itself (squared). These values are then summed up and divided by the number of mid-ordinates (or samples) n. In our case we sampled every millisecond or 1000 times per second to retrieve the mean value of the RMS voltage expression. Finally, we calculate the square root of the mean value to get our voltage. Because the ACS712 uses 5 volts as Vcc the result must be multiplied by 5/1024 to get the V_{rms} (see expression 2).

$$V_{RMS} = \sqrt{\frac{\sum(Sample_i - 512)^2}{n}} * \frac{5}{1024} \tag{2}$$

According to the datasheet our ACS712 is a $20A$ meter and has a sensitivity of $100mV$ per Ampere. In order to calculate the amperage (denoted as A_{RMS}) in our setup we need to divide the result from expression 2 by 100 (see expression 3). Finally, to get the number of Joules consumed we need to know the wattage which can calculated by multiplying the A_{RMS} by the voltage that comes out of our power socket which is $230V$ in our case. The A_{RMS}, wattage and total number of Joules are printed to the LCD display.

$$A_{RMS} = V_{RMS}/100 \tag{3}$$

It proofed difficult to cancel out the interference that was generated in the environment probably caused by wifi and devices connecting constantly to wireless networks. After turning all devices off that were in the near vicinity of the current sensor we still had problems measuring sometimes up to 0.8 Ampere without any power consumption. Moving the current meter by just a few millimetres would eventually result in a 0 Ampere measurement with a device (light bulb or vacuum cleaner) connected and not turned on. There is room for improvement to this setup (see section 8) but the results were accurate enough to validate our method as described in more detail in the next section.

6 Case study

Two small pieces of code were used to test the validity of our method. We wrote the code and calculated the energy consumption using our trace generator and energy analysis tool. To run the code on an Arduino it had to be slightly adapted but changes were limited to the definition of the external components. The Arduino requires two functions, a *setup* and a *loop* function, where the loop function is running continuously unless terminated. To run the code through our trace generator we renamed the *setup* function to *main* and kept the *loop* function as is. The calls to external components were kept in the *loop* function to keep the simulation as realistic as possible.

Before running the tests the cost model had to be adapted to the costs of the Arduino. To do this a speedtest was created measuring the time in microseconds for each instruction and some important functions. The data was then converted to cycles based on the clock speed of 16Mhz. Although the results are reproducible the number of microseconds were measured by repeating the same instruction and averaging the time required for a single instruction. Under normal circumstances this would not be a good approach but in case of the Arduino Uno there is no caching involved making the measurements accurate enough for our test.

The first test was run with a nested loop of which the iterator was defined by an array of values. These values were not given as program argument but kept as an array in code. Due to the complexity of the array this code was not optimized by the compiler which meant that the value of the iterator was calculated at

runtime. This made it a bit easier to run tests and adapt the code later on to make it suitable for the Arduino. As can be seen in listing 10 the goal here is to

```
size_t n = (sizeof(d) / sizeof(d[0]));
for (int x = 0; x < n; x++) {
  for (int i = 0; i < d[x]; i++) {
    int mod = i % 5;
    if (mod > 0 && mod < 4) {
      turnOff(); // component function
      delay(5000);
    }
    else {
      turnOn(); // component function
      int x = Fibonacci(1000);
    }
  }
}
turnOff();
```

Listing 10: *Fibonacci example in loop function*

turn the device on for a period of time and keep it on for the duration of the Fibonacci calculation. By using a nested loop using free variables to define the number of iterations the code would execute in a different way for each input parameter and not be optimized to a more simple form. To increase the time delay caused by the Fibonacci function an extra delay function was added at the end of the Fibonacci function increasing the total algorithm time delay with 5 seconds.

The second example (see listing 11) was created with input parameters to simulate branching by a free variable. All the code does is wait for a string of input variables where the character *a* would turn the device on and any other character, say *b* would turn it off. Here also a delay was added at the end of the *turnOn* and *turnOff* functions to increase the duration of the power consumption. The advantage with the *String test* example is that when the string is consumed another string can be offered and the execution will continue and not terminate like with the *Fibonacci* example.

In table 1 the measurements are shown that were measured by the Joules meter and the calculation that was done by the analysis tool for both code examples on the light bulb. The time shown is not the total execution time but the time power was drawn. Basically this means that the wattage multiplied by the time equals the number of Joules. In the measurements this is not exact because the wattage here is an average and calculated over time. The difference between the number of Joules and time with the *Fibonacci* example is greater than the *String test* example. The complexity of the code has a smaller effect on the execution time than expected in the analysis which explains the difference. This indicated

```
if (Serial.available()) {
  input = Serial.read();
  if (input == 97) // a
    turnOn();
  else           // b
    turnOff();
}
```

Listing 11: *String test: uses an input string to determine consumption*

that despite having a complex algorithm like a Fibonacci calculation, the effects are negligible when calculating energy consumption of external components with the ones we used. The wattage for the Fibonacci and the string test are both

	Measurements			Analysis			Compensated	
Code	**Time**	**Joules**	**Watt**	**Time**	**Joules**	**Watt**	**Joules**	**Watt**
Fibonacci	533	38526	74	544	40789	75	40256	74
String test	102	7265	71	102	7650	75	7242	71

Table 1: *Light bulb measurements compared to analysis*

different from the analysis. This is because the light bulb uses 75W on paper but in reality fluctuates between 65 and 75 Watt due to environmental conditions like heat and ageing of the filament. When we compensate the analysis by changing the wattage annotation from 75 to 74 in the Fibonacci example the difference in Joules is smaller. Note that the difference in the Fibonacci example is only 1730 Joules after compensation which is 3.24 Joules per second.

In total the difference between the measured and analysed results were approximately 4.2% with the Fibonacci example and 0.3% in the String test in our light bulb test after compensation.

The same tests were executed for the vacuum cleaner and here the differences are larger in terms of absolute numbers. This was as expected because a motor does not draw a constant amount of power like a light bulb does. In order to normalize the power consumption the delays that influenced the duration of the component state (*on*/*off* frequency) were increased from 5 to 20 seconds. This is also why the values for time are different in table 2. Again here the difference seems large when looking at the number of Joules even for the compensated values. The difference in the number of seconds that power is being drawn is also much larger. However, when looking at the total time the difference is 34 seconds on a total time of more than 31 minutes of time dependent energy

	Measurements			Analysis			Compensated	
Code	**Time**	**Joules**	**Watt**	**Time**	**Joules**	**Watt**	**Joules**	**Watt**
Fibonacci	1.914	1132000	591	1880	1034000	550	1111080	590
String test	695	409120	590	680	374000	550	401200	590

Table 2: *Vacuum cleaner measurements compared to analysis*

consumption of approximately 590W. When factoring in that a motor requires more power upon start, larger differences are to be expected and are still within a tolerable margin. The total difference for both the Fibonacci and the String test is no more than 2% which is smaller in relative terms than the results of the light bulb.

Because the *String test* seems to be more accurate we decided to take a closer look in an attempt to find out what might cause these differences. The results of this test are shown in table 3 where an a would turn the vacuum cleaner on for 20 seconds and b turns it off for 8 seconds. The time shown in table 3 for static analysis is equal to the number of occurrences of a (device is turned on) multiplied by 20 seconds. Multiplying this time with the wattage gives the number of Joules.

The first thing we did is calculate the difference between the Joules measured by the device and the results created by the static analysis tool (see table 4). We then calculated the frequency ratio by taking the number of consecutive a's in the string and divided these by the total number of characters in the string. These frequency ratio's show that the difference in time-dependent energy consumption increases when the frequency of switching the vacuum cleaner on and off is increased (see table 4). Thus decreasing the frequency had a positive influence on the accuracy of our method in case of a device with a motor like our vacuum cleaner. The data indicates that the hybrid approach gives an

	Static Analysis			*Measurements*			
Input values	**Time**	**Joules**	**Watt**	**Time**	**Joules**	**Watt**	**Diff (J)**
baabbaabaabbbaaaababab	240	141600	590	243	144831	596	3231
bbbaaabbaaabbbbaaaab	440	259600	590	448	264699	591	1868
bababbbaaabbbbaaabbabbaaab	680	401200	590	695	409120	589	2821

Table 3: *Vacuum cleaner String test cumulative results shown in three steps*

Input values	**Diff (J)**	**Frequency ratio**
baabbaabaabbbaaaababab	3231	0.27
bbbaaabbaaabbbbaaaab	1868	0.15
bababbbaaabbbbaaabbabbaaab	2821	0.23

Table 4: *Differences between between measured and statically analysed in Joules and frequency ratio on vacuum cleaner with* String test.

accurate indication how much energy will be consumed without using an expensive measuring setup. Depending on the hardware that is being measured a certain margin for error has to be included. The results with the light bulb show that more accurate results are achieved when the external components draws a more constant amount of energy. The test results with the vacuum cleaner show that less frequent component state changes with devices like pumps or motors will result in a more accurate result. In our method we measure the cost of an instruction and although this does increase accuracy the *Fibonacci* test cases showed a larger difference than the *String Test*. Increasing the number of calculations by increasing the parameter only increased the execution time by a few microseconds making it negligible compared to the built-in delays of a few seconds.

7 Related work

The developed approach as described in this paper does not stand on its own and has roots in previous research some of it closely related. In [6, 9] an analysis is performed on an illustration language called ECA. ECA is constructed with a simple grammar in mind to allow formal reasoning. The problem with the ECA language is that it cannot be compiled at this point and is thus not very useful in real world practices. Furthermore, most hardware controllers are written in languages like ANSI C. This research expands on this research and can be seen as an attempt to create a practical approach supporting multiple languages and achieve a higher precision in terms of timing by analysing a target independent, lower-level intermediate representation language called LLVM IR.

Most of the current research focuses on design patterns and high-level languages trying to define energy-efficient algorithms [1, 3, 13]. There is work on software design in a modular way with focus on user behaviour [5]. Other work takes the approach of programs divided into phases that describe a similar behaviour and proposes design optimisations that must improve code and become more energy-efficient [14, 15].

More related work focuses on energy consumption of instructions on CPU architectures [7, 11], Cost analysis [2] and Worst-Case Energy consumption [8]. In some cases analysis is performed on an intermediate representation [4] or focusses on each individual instruction on XMOS ISA-level as in [11]. The major difference with this research is the focus on external hardware components. We explicitly do not focus on the consumption of the machines or controller the software itself is running on but on the hardware it controls.

To calculate energy consumption using only static analysis we need to determine the path through the program. In [8] a technique is presented to estimate the worst-case energy consumption on micro-architectures. They use Integer Linear Programming (ILP) with constraints to estimate the WCET but still require the user to provide the maximum number of iterations for loops. In [10] an *implicit* path enumeration model is presented to find the worst-case path through the program and implements this solution in a program called *cinderella*. They extract linear constraints from the control flow graph (CFG) in order to find the worst-case execution time of a program. The result is still a worst-case path and loop bounds must be provided by the user of *cinderella*. In our approach we use discretization of input parameters to simulate real world use and generate a program path through partial execution.

8 Future work

With our current approach, each component state needs to be annotated. In case of many different hardware components and state changes the number of annotations becomes large and difficult to maintain. An improvement would be to be able to setup a range of states that can be passed by a variable. For instance, a screen brightness setting would normally be a component function call with a parameter. The analysis tool would need to evaluate these kinds of parameters and know what the effect on the component state should be. For instance we could add an operator that needs to be applied on the parameter (i.e. increment, multiply or negate in case of a boolean).

The data showed that increased code complexity did not increase the execution time by a large factor. Further testing is required with algorithms that have a significant impact on the execution time and with it the energy consumption. To increase the accuracy in terms of timing the analysis can be done on the target dependent assembly level. This would require a backend compiler for the specific target architecture rendering the method less flexible but more accurate. This approach could also be used to increase timing accuracy by measuring the time an operator requires on a certain architecture. These result can then be incorporated by adding a target specific cost model as program argument to the analysis tool.

When it comes to accuracy in terms of timing the speed of the CPU and the number of cycles of each operator depends on the target architecture. For the

Arduino Uno we used speed test which is a rather crude method to calculate the number of microseconds required for each instruction. To retrieve a more accurate result the read and write speed to memory (cache or otherwise) should be taken into account. Also testing should be done for a longer period of time under heterogeneous conditions (temperature, electrical interference etc) to achieve higher precision.

In our test we encountered discrepancies in the measured amperage. This was due to the crude setup and noise generated in the test environment interfering with the current meter. Creating a setup that does not suffer from outside interferences is more costly but should provide more accurate results. With a solid measuring setup more tests on different devices should indicate how hardware components like motors consume energy and what the effect of heat collection will have on the components energy consumption. This kind of research can then be incorporated into the resource consumption formula to create more accurate results.

9 Conclusion

This paper presented a hybrid method to analyse resource consumption that is language and target independent by combining dynamic and static analysis. Hardware was abstracted and added as annotations that include *time-dependent* and *time-independent* resource consumption. The annotated software was then compiled to LLVM IR and transformed into a deterministic executable program. By executing this program with (discretised) input values a trace was generated that represented a realistic path through the program eliminating the need for path analysis and branch prediction. The generated trace was then used to statically analyse the resource consumption of the (external) hardware components on LLVM IR and produced accurate results that were validated in a case study (see section 6). This approach allows for a more cost effective development of energy efficient hybrid systems that can be easily integrated into existing development pipelines.

Dynamic analysis gives a more realistic result in contrast with static analysis but requires a test setup that is difficult to integrate in an incremental development process. Static analysis requires complex predictions of conditions, branches and program state with inaccurate (worst-case) results. By combining both methods we were able to utilize the advantages of both approaches. Now every developer can analyse and test the energy consumption of their software with high accuracy and without complex test setups at any given moment. The presented method in this paper can be used with every programming language that has a frontend compiler to LLVM IR and is target independent.

With the LLVM Framework we had the ability to transform and traverse the code and execute it using *lli*, a LLVM bitcode interpreter. By getting rid of all

code that is not relevant to create a trace we could execute the code creating a program trace without the use of external hardware components, eliminating one of the disadvantages of dynamic analysis. With the resulting program trace there was no need for path analysis and condition checking speeding up the process of static analysis without loosing accuracy in terms of timing. The program trace eliminated the need for relatively complicated program path analysis, condition prediction and loop analysis.

We validated our method by testing the energy consumption of a light bulb and a vacuum cleaner. To increase measuring accuracy a Joules meter was designed using a current meter (see section 5). The current meter proved to be sensitive to interference creating some noise in the measurements. Despite issues with the current meter we still managed to create measurements that came close to our calculated results. We measured differences between 0.3 and 4.2 percent for the ligh bulb and less than 2 percent for the vacuum cleaner (see section 6). Although, further testing is required we can say that the method is accurate enough to analyse the consumption of external components that draw a constant amount of power. Tests with the vacuum cleaner indicated that hardware with a motor (or a pump) show less accurate results in terms of absolute numbers. We did find that when the frequency of changes in the component state are reduced the results are more accurate.

The effects of complex algorithms on the total resource consumption was negligible in our test. We used a Fibonacci function to increase resource consumption by running the function before turning the device off. We had to add a delay of a few seconds in the Fibonacci function to notice any effects on the total resource consumption. In future work more time consuming algorithms should be tested to measure the effects of complexity of algorithms on resource consumption.

References

1. Albers, S.: Energy-efficient algorithms. Communications of the ACM **53**(5), 86–96 (2010)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Cost relation systems: A language-independent target language for cost analysis. Electronic Notes in Theoretical Computer Science **248**, 31–46 (2009)
3. Boukerche, A., Nikoletseas, S.: Energy-efficient algorithms. Algorithms and Protocols for Wireless Sensor Networks **62**, 437 (2008)
4. Brandolese, C., Corbetta, S., Fornaciari, W.: Software energy estimation based on statistical characterization of intermediate compilation code. In: Low Power Electronics and Design (ISLPED) 2011 International Symposium on. pp. 333–338. IEEE (2011)
5. te Brinke, S., Malakuti, S., Bockisch, C., Bergmans, L., Akşit, M.: A design method for modular energy-aware software. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. pp. 1180–1182. ACM (2013)
6. van Gastel, B.E.: Assessing sustainability of software: analysing correctness, memory and energy consumption. Ph.D. thesis, Open University, The Netherlands (2016)
7. Grech, N., Georgiou, K., Pallister, J., Kerrison, S., Eder, K.: Static energy consumption analysis of llvm ir programs. Computing Research Repository, arXiv pp. 1–12 (2014)
8. Jayaseelan, R., Mitra, T., Li, X.: Estimating the worst-case energy consumption of embedded software. In: Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE. pp. 81–90. IEEE (2006)
9. Kersten, R., Toldin, P.P., van Gastel, B., van Eekelen, M.: A hoare logic for energy consumption analysis. In: Dal Lago, U., Peña, R. (eds.) Foundational and Practical Aspects of Resource Analysis. pp. 93–109. Springer International Publishing, Cham (2014)
10. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: ACM SIGPLAN Notices. vol. 30, pp. 88–98. ACM (1995)
11. Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M.V., Eder, K.: Energy consumption analysis of programs based on xmos isa-level models. In: International Symposium on Logic-Based Program Synthesis and Transformation. pp. 72–90. Springer (2013)
12. LLVM: The llvm compiler infrastructure project, https://llvm.org
13. Noureddine, A., Rajan, A.: Optimising energy consumption of design patterns. In: Proceedings of the 37th International Conference on Software Engineering-Volume 2. pp. 623–626. IEEE Press (2015)
14. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: Enerj: Approximate data types for safe and general low-power computation. In: ACM SIGPLAN Notices. vol. 46, pp. 164–174. ACM (2011)
15. Saxe, E.: Power-efficient software. Commun. ACM **53**(2), 44–48 (Feb 2010). https://doi.org/10.1145/1646353.1646370, http://doi.acm.org/10.1145/1646353.1646370