Products go Green: Worst-Case Energy Consumption in Software Product Lines

Marco Couto
HASLab/INESC TEC
Universidade do Minho, Portugal
marco.l.couto@inesctec.pt

Paulo Borba
CIn
Univ. Federal de Pernambuco, Brazil
phmb@cin.ufpe.br

Jácome Cunha
NOVA LINCS, DI, FCT
Univ. Nova de Lisboa, Portugal
jacome@fct.unl.pt

João Paulo Fernandes
Release/LISP, CISUC
Universidade de Coimbra, Portugal
jpf@dei.uc.pt

Rui Pereira
HASLab/INESC TEC
Universidade do Minho, Portugal
ruipereira@di.uminho.pt

João Saraiva
HASLab/INESC TEC
Universidade do Minho, Portugal
saraiva@di.uminho.pt

ABSTRACT

The optimization of software to be (more) energy efficient is becoming a major concern for the software industry. Although several techniques have been presented to measure energy consumption for software, none has addressed software product lines (SPLs). Thus, to measure energy consumption of a SPL, the products must be generated and measured individually, which is too costly.

In this paper, we present a technique and a prototype tool to statically estimate the worst case energy consumption for SPL. The goal is to provide developers with techniques and tools to reason about the energy consumption of all products in a SPL, without having to produce, run and measure the energy in all of them.

Our technique combines static program analysis techniques and worst case execution time prediction with energy consumption analysis. This technique analyzes all products in a feature-sensitive manner, that is, a feature used in several products is analyzed only once, while the energy consumption is estimated once per product.

We implemented our technique in a tool called *Serapis*. We did a preliminary evaluation using a product line for image processing implemented in C. Our experiments considered 7 products from such line and our initial results show that the tool was able to estimate the worst-case energy consumption with a mean error percentage of 9.4% and standard deviation of 6.2% when compared with the energy measured when running the products.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; **Software performance**; **Software product lines**;

ACM Reference format:
Marco Couto, Paulo Borba, Jácome Cunha, João Paulo Fernandes, Rui Pereira, and João Saraiva. 2017. Products go Green: Worst-Case Energy Consumption in Software Product Lines. In *Proceedings of SPLC '17, Sevilla, Spain, September 25-29, 2017,* 10 pages.
DOI: 10.1145/3106195.3106214

1 INTRODUCTION

The widespread use of non-wired devices and the advent of the internet-of-things, where most consumer electronics are (powerful) computing devices, is changing the way software engineers develop their software. Software has to run in a variety of devices and energy consumption is becoming *the* bottleneck in terms of software performance. Software Product Lines (SPL) have emerged as an important software engineering discipline allowing the development of software that shares a common set of *features*. Thus, SPLs are particularly suitable to develop software where individual *products* target specific computing architectures/devices, while sharing common software *features*.

Although several techniques have been proposed to measure energy in different scopes [8, 11, 16], there is no proposed technique to address this problem in SPLs. A *brute-force* approach would be to generate all *products* within a SPL and use common techniques to measure each *product* energy consumption. This would, however, be too costly, since, for each *product*, it would be necessary to instrument it and run measurement tools on it. This can easily become unfeasible depending on the size of the SPL.

In this paper we present techniques to reason about energy consumption in the context of software product lines based on conditional compilation. Our goal is to providetechniques and tools to software developers, allowing them to identify (non) green *products* and/or *features* in a SPL. Moreover, we wish to avoid a *brute-force* approach where all *products* are first generated, so that their energy consumption is monitored and analyzed at runtime. As a consequence, common *features* shared by several *products* would be forced to be repeatedly analyzed.

We aim to analyze the SPL in a feature-sensitive manner, i.e., a *feature* that is used in several *products* is analyzed only once. In order to do so, we use SPL static analysis techniques [2] that work on a control-flow graph representing a set of programs (the SPL), and not just one. This analysis is used to compute 2 things from the source code: *i)* energy related properties, such as hardware components usage information, and *ii)* dataflow information, such as loop upper bounds.

The computed properties are then used as input by the next step of our technique. This step is based on the Worst-Case Execution Time (WCET) prediction technique, where instead of execution time we compute the energy consumption per *product*. This is the only step where the analysis is product-oriented. We use a constraint solver to estimate the consumption, where, for each *product*, we generate a set of constraints. Note that for each shared *feature* only one set of constraints is generated. However, due to the nature of a constraint solver, the constraints of a *feature* are calculated for every product including it. In fact, this is the only part of our approach where we recalculate information for shared *features*. We call this new technique the *Worst-Case Energy Consumption* (WCEC).

Our WCEC analysis technique for *products* in a SPL implies the existence of an energy consumption model. This model abstractly describes the hardware where the *products* will run, and the amount of energy needed by different types of instructions to execute. Using such model along with the previously computed properties, our technique computes the energy consumption profile for each *product* and *feature*. Thus, it allows the generation of the most energy efficient *product* that includes/excludes a set of given *features*.

Moreover, we have implemented WCEC in a prototype tool, which given the SPL source code statically estimates the energy consumption of its *products*. This tool extends a C Intermediate Language (CIL) front-end with SPL primitives, and the WCEC analysis. To assess the precision of our technique we considered a C-based SPL included in the San Diego Vision Benchmark Suite [24]. Our preliminary results show that the energy predicted by our tool for each *product* is always, as expected, an overestimation of the real consumption, diverging on average by 9.4%.

The remaining of this paper is structured as follows: Section 2 provides a brief introduction to static analysis in the context of SPL. Section 3 presents in detail our contribution: the worst-case energy consumption analysis. Section 4 describes the architecture of the WCEC prototype tool we developed. In Section 5 we show the results of using the tool in a real SPL. Finally, Section 6 describes related work and we include the conclusion of our work in Section 7.

2 STATIC ANALYSIS IN SPL'S

The goal of our work is to develop a methodology for statically predicting the energy consumption of *products* in a SPL. In order to better explain our approach, we first need to explain some introductory concepts that we based our approach on.

In this section, we will explain how static analysis can be achieved in SPLs. We start by explaining SPLs concepts in Section 2.1. Then, we will explain the static program analysis concepts in Section 2.2, and finally we will present and explain the chosen technique for integrating all the previously stated concepts in order to achieve static analysis in SPLs Section 2.3. The concepts included in this section are mostly inspired from the contents of [2].

2.1 Basic Concepts of Software Product Lines

Research on Software Product Lines has been focused on software engineering methods and techniques designed to manage variability in a software system, in a normalized manner [1, 2]. Using SPLs allows us to create a collection of different software solutions from reusable assets in the same software system (i.e., code fragments, visual assets, etc.), using the same means of production.

In order to illustrate the essential aspects of SPLs, let us consider the purchase of a car. In such a situation, once the particular car model is choosen, it is often required to configure it according to preferences and budget. Configuration options are diverse, and often involve choosing the engine type, entertainment system, etc..

Imagine that the following (simplified) configurations are possible for a given car:

- turbo engines are not available with the basic version;
- turbo engine is mandatory when choosing air-conditioning.

In the context of SPLs, the *Basic* version, *Turbo* or air-conditioning (*Air*) are often referred to as *features*.

Thus, a *product* (e.g., a concrete car that respects the configurations which are possible) is characterized by the set of *features* which it includes/implements (e.g., $\{Car, Basic, Air, Turbo\}$). This set of *features* is called the *product configuration*, and can be used to generate several *products* of the same kind.

The number of possible *product configurations* may sometimes be restricted by a so-called *feature model*. This model is a propositional logic formula, which deals with situations such as exclusive *features*, and *features* that may imply the inclusion of other *features*.

For instance, considering our running example and the *feature model* $\Psi = Car \wedge (Basic \Leftrightarrow \neg Turbo) \wedge (Air \Rightarrow Turbo)$, the complete set of *product configurations* is:

```
{{Car, Basic}, {Car, Turbo}, {Car, Air, Turbo}}
```

One way to define if a code block belongs to a specific feature is by using *conditional compilation*. This technique is based on associating to the code block a pre-processor instruction, an `#ifdef` Φ, where Φ is a propositional logic formula over *feature names*:

$$\Phi ::= f \in \mathbb{F} | \neg\Phi | \Phi \wedge \Phi | \Phi \vee \Phi$$

Where f is a *feature* name, drawn from a finite alphabet of *feature* names $\mathbb{F}$. This allows to indicates which *features* include that code block, or which one must exclude it.

2.2 Static Dataflow Analysis concepts

In order to briefly review classic static program analysis concepts, we will provide illustrations based on the following code example:

```
void m() {
  int i = 0;  int x = 0;
  x = input();      //1..100
  #ifdef (A) x*=2;
  #ifdef (B) x--;
  while(i < x){ i++; }
  }
```

Listing 1: Example of a SPL method

In this example, conditional compilation primitives are used to define a SPL where two *features*, A and B, are available. If A is present in a particular *product*, the value of x doubles after obtaining the input; if B is present, the value of x is decremented.

Every classic static dataflow analysis consists of three components: a) a *control-flow graph*, representing the connection between instructions (and on which the analysis is performed); b) a *lattice*, representing the values of interest for the analysis; and c) *transfer functions*, responsible for simulating the execution of the program represented by the *control-flow graph*.

These components, are the input to the *fixed-point computation function*. This function is responsible for calculating a *lattice* value for each node of the *control-flow graph*, which is the fixed point of the transfer functions at that point.

Control Flow Graph (CFG): This component is an abstraction of the program given as input. A CFG is a directed graph, with the statements of the program to be analyzed as nodes, and the edges representing the flow of control. An edge can be assigned a boolean expression, which is the condition to be verified in order for the program to follow that flow. Figure 1 represents the CFG for the *product* derived from the example in the Listing 1, when only *feature* A is included.

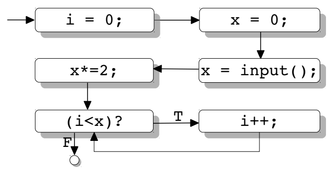


Figure 1: Control-Flow Graph for the example in Listing 1

Lattice: When performing static dataflow analysis, the calculated information is arranged in a *lattice*, $L = (D, \sqsubseteq)$, where D is a set of elements and $\sqsubseteq$ is a partial-order on the elements.

Each element represents information relevant for the analysis to be performed. For example, if the goal of the analysis is to check the signal of a variable (called a *sign analysis*) the element "+" indicates that a value is always *positive*, while the element "0/+" represents *zero or positive*. Figure 2 represents a *lattice* for the *sign analysis*.

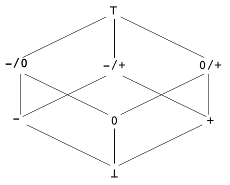


Figure 2: Lattice for Sign Analysis

There are two special elements in the lattice: $\bot$, at the bottom of the *lattice*, usually meaning the node was not yet analyzed, and $\top$, at the top of the *lattice*, usually meaning it is impossible to determine the *lattice* element for that node.

The partial order induces a *least upper bound operator*, represented by $\sqcup$. This operator allows information to be combined during the analysis, when a node has more than one entry point (for example when there is a *loop* or an *if*). For instance, in *sign analysis* we have: $\bot \sqcup 0 = 0$, $0 \sqcup + = 0/+$, and $- \sqcup 0/+ = \top$.

Transfer/Update Functions: A *transfer function* is a monotone function which simulates the execution of a statement (with respect to what is being analyzed). Each statement has an associated transfer function which receives one or more *lattice* elements and uses it to compute a new element associated to the statement. The input elements for the function are the *lattice* elements associated to the predecessors or successors relevant for the analysis. For example, if we want to perform the *sign analysis* for the example in Figure 1 the *transfer function* for the statement `x--` would be:

$$f_{\mathtt{x--}}(l) = \begin{cases} \top & : l \in \{-/+, 0/+, \top\} \\ - & : l \in \{-, -/0, 0\} \\ 0/+ & : l = + \\ \bot & : l = \bot \end{cases}$$

In this case, l is the *lattice* element of the only predecessor of `x--`. If that element is for example `0/-`, it means that before entering the statement `x--` the value of `x` was either zero or a negative value. Given those possibilities, decrementing `x` will only result in a negative value, as the function shows. In order to assure that the analysis is well-defined, all the transfer functions must obey the monotony property.

Once every transfer function is defined, the first step of the analysis will be to transform the CFG into a *whole-program transfer function*, T, which contains the *transfer functions* for all points of the program to be analyzed. Figure 3a shows the *whole-program transfer function* for *sign analysis* of the example in Figure 1.

The last step of static analysis will then be to calculate the *fixed point* of the *whole-program transfer function*. Figure 3b shows the result of such computation for the example in Figure 1, using function T. This analysis enables us to determine the sign of the variable `x` in every statement of the program.

2.3 Static Analysis in Software Product Lines

Static analysis, as explained in Section 2.2, can only be used in programs with no included variability. In a SPL, there is a need to make the analysis *feature* aware, so it can compute the results for all *products* at once.

As explained in [2], there are different ways to achieve feature-aware static analysis. For both efficiency and ease of implementation, in our context we have chosen to adopt *Simultaneous Feature-Sensitive Analysis*. In the following paragraphs, we will explain what this technique consists of and how it combines static dataflow analysis with *feature* awareness.

Simultaneous Feature-Sensitive Analysis: A SPL can be analyzed by building all possible *products* and analyzing them individually. However, this is not desirable as a SPL with a substantial number of *features* will give rise to a considerable number of *products*. It is possible to avoid building all *produts* by making the dataflow analysis *feature* sensitive. In order for this, there are a few changes that need to be made to the static dataflow analysis components.

The CFG needs to be changed in a way that it is possible to know if a statement is *feature* dependent or not. In order to do this, each node of the CFG has assigned the set of *features* which implement the corresponding statement. Each node will now have a pair $(S, \mathcal{F})$, where S (e.g., `x*=2`) is the statement and $\mathcal{F}$ (e.g., $[\![A]\!]$) is the list of *features* where S is implemented. Unconditional statements will have the list of all *configurations*, represented as $[\![TRUE]\!]$.

The *lattice* will also change, since it is now necessary to know which *products* are affected by the execution of every instruction. This can be achieved using a so-called *lifted-lattice*, which will basically maintain for every node one *lattice* element per valid *product configuration*. This way it is possible to know the result of the analysis on all *products*. Similarly, the *transfer functions* will also be lifted, i.e., they will only be applied to the *product configurations* which include the statement being analyzed.

Figure 4 shows the final iteration of the fixed point computation, using the *Simultaneous Feature-Sensitive* approach to perform the *sign analysis* in the example of Listing 1. The result here is expressed as an annotated CFG. At the exit of each node there is a lifted *lattice* with an element associated to each possible *product* of the SPL, where each element is the result of the *transfer function* associated to the node.

$$T\begin{pmatrix}a\\b\\c\\d\\e\\f\end{pmatrix} = \begin{pmatrix} f_{\texttt{i=0}}(\bot)\\ f_{\texttt{x=0}}(a)\\ f_{\texttt{x=input}}(b) \\ f_{\texttt{x*=2}}(c)\\ (d\sqcup f)\\ f_{\texttt{i--}}(e)\end{pmatrix}$$

(a) Whole-program Transfer Function

a	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
b	$\bot$	0	0	0	0	0
c	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$
d	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
e	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$
f	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$
	T^0	T^1	T^2	T^3	$T^4 = T^5$	

(b) Fixed-point Computation

Figure 3: Whole-program transfer function and sign analysis (as fixed point computation) for the example in Figure 1

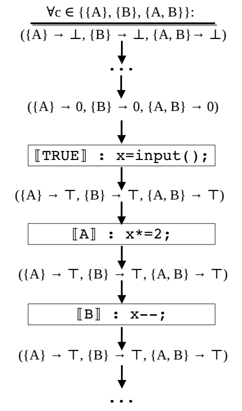

∀c ∈ {{A}, {B}, {A, B}}:
({A} → ⊥, {B} → ⊥, {A, B} → ⊥)
. . .
({A} → 0, {B} → 0, {A, B} → 0)
⟦TRUE⟧ : x=input();
({A} → ⊤, {B} → ⊤, {A, B} → ⊤)
⟦A⟧ : x*=2;
({A} → ⊤, {B} → ⊤, {A, B} → ⊤)
⟦B⟧ : x--;
({A} → ⊤, {B} → ⊤, {A, B} → ⊤)
. . .

Figure 4: Result of *sign analysis* performed for the example in Figure 1 using the *Simultaneous Feature-Sensitive* approach

3 STATIC ENERGY ANALYSIS IN SPLS

In Section 2 we explained how static analysis in SPLs can be performed by combining typical state-of-the-art static analysis techniques with the components that characterize a SPL, such as features and feature models.

However, all the presented concepts and techniques were for the purpose of static analysis: predicting whether a property is true or maintained after a certain instruction. In other words, static analysis, as presented until now, enables the possibility of predicting the behavior of all products in a SPL by analyzing their code.

In order to achieve static energy consumption analysis in a SPL it is necessary not only to determine such behavior, but also to understand how it can affect energy consumption. A more thorough analysis, similar to worst case execution time (WCET) analysis, is needed. In such an analysis, the goal is to give an upper bound for each instruction, determine how they all are related, and calculate an estimate for the worst case scenario.

We present, in this section, our strategy for determining an energy consumption worst case scenario based on the classic WCET analysis [25]. This strategy is composed of 4 phases:

A. first, use fixed-point computation to gather information about how instructions' execution influences the hardware energy consumption behavior;
B. use a data-flow analysis to determine loop upper bounds;
C. at each program point, an energy model calculates an energy-bound for every product of the SPL;
D. finally, use all the information from previous steps, a constraint solver computes a global energy-bound for all the products of the SPL.

3.1 Static Prediction of Energy Behavior

In classic execution time prediction, the first step towards determining the WCET is to analyze how the processor will behave when executing a certain statement [25]. This allows us to determine how the processor will behave while also depending on what statement(s) was(were) executed before. This is called the *Processor Behavior Analysis* step.

The goal is to create an abstract model of such behavior, with all the processor states which can affect energy consumption. In other words, to create a finite *lattice* where the elements are those states, and the necessary conditions to switch state are described by monotone *transfer functions*.

Having such an abstract model for the processor's energy behavior, all the properties from static dataflow analysis (as presented before) hold. This was already proven to work before [14]. For our purpose, we want to make it possible to use this approach to model every hardware component, in every possible state, which influences the energy consumption.

To fit static program analysis requirements, each component's behavior must be independent of all others. Nevertheless, the behavior analysis must be done only once. We need to define a prediction model, $\mathbb{P}$, able to determine how the hardware components behave when executing a given instruction. The model $\mathbb{P}$ consists of a set of *transfer functions*. The input of the prediction model $\mathbb{P}$ is a list of n *lattice* elements, $\mathbb{S}$, one for each hardware component, and the output is $\mathbb{S}'$, corresponding to a list of new *lattice* elements. When analyzing a statement, the elements in $\mathbb{S}'$ represents the state of the components **after** the statement is executed.

As shown in **??**, the analysis in a SPL must be extensible to every product. This requires two things: (1) for every node/program point there must be a list of *lattice* elements for every possible product, and (2) for a given product p the corresponding list of *lattice* elements must be updated (using the *transfer functions*) if and only if the instruction in the node is included in product p.

The result of an analysis such as this one is similar to the one presented in Figure 4. However, each node will have associated a map between product configurations and a list of n *lattice* elements (for n hardware components), instead of only one *lattice* element.

3.2 Data-Flow Analysis

In the context of WCET, data-flow analysis is used for gathering information about the possible set of values for a variable at various program points [25]. This is also a key component for our approach, since it enables us to determine the variables' values before/after the execution of each iteration of a loop. We can use such information to test the loop condition after every iteration, and determine the maximum number of iterations: the loop upper bound.

In WCET analysis all loops must have an upper bound. In fact, determining a loop bound is crucial to increase the accuracy of the analysis. However, it is often impossible to accurately determine such a bound. In order to address this problem, usually additional information is provided by the programmer: a range of values that input variables may have.

Assuming that such information is provided, we developed a technique capable of determining the upper bound of any loop. This technique is the result of combining static program analysis to determine the variables' values at each program point [14], and abstract execution to automatically derive loop bounds [5].

Figure 5 expresses the result of our technique in Example 1 at every program point until the first iteration of the loop concludes. We can see that the analysis is feature sensitive, i.e., the propagation of a variable's abstract value in an assignment is only considered for products which include such assignment.

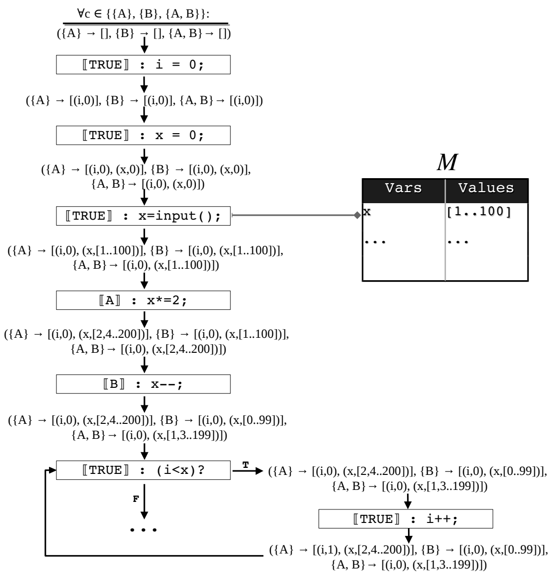

∀c ∈ {{A}, {B}, {A, B}}:
({A} → [], {B} → [], {A, B} → [])
⟦TRUE⟧ : i = 0;
({A} → [(i,0)], {B} → [(i,0)], {A, B} → [(i,0)])
⟦TRUE⟧ : x = 0;
({A} → [(i,0), (x,0)], {B} → [(i,0), (x,0)], {A, B} → [(i,0), (x,0)])
⟦TRUE⟧ : x=input();
({A} → [(i,0), (x,[1..100])], {B} → [(i,0), (x,[1..100])], {A, B} → [(i,0), (x,[1..100])])
⟦A⟧ : x*=2;
({A} → [(i,0), (x,[2,4..200])], {B} → [(i,0), (x,[1..100])], {A, B} → [(i,0), (x,[2,4..200])])
⟦B⟧ : x--;
({A} → [(i,0), (x,[2,4..200])], {B} → [(i,0), (x,[0..99])], {A, B} → [(i,0), (x,[1,3..199])])
⟦TRUE⟧ : (i<x)?
T
F
...
({A} → [(i,0), (x,[2,4..200])], {B} → [(i,0), (x,[0..99])], {A, B} → [(i,0), (x,[1,3..199])])
⟦TRUE⟧ : i++;
({A} → [(i,1), (x,[2,4..200])], {B} → [(i,0), (x,[0..99])], {A, B} → [(i,0), (x,[1,3..199])])
M
Vars
Values
x
[1..100]
...
...

Figure 5: Example 1 data-flow analysis (loop's first iteration.)

Our approach follows the forward analysis principle [14]. It begins at the program's first instruction, and once it reaches an assignment the abstract value for the variable is updated. The result is propagated to the following instructions, and so on. Considering, for example, that the instruction being analyzed is an assignment of the value 5 to a variable *a*. The abstract value for *a* will be 5, and the successor instructions will consider that value for variable *a*.

Our technique requires a map structure, *M*, which relates variables and a range of possible values for them. Every time an instruction is found that assigns an input value to a variable, the map structure *M* is consulted and the variable gets an abstract value representing its range of possible values. If there is no entry in the map, the variable gets the abstract value $\top$, which will be considered as the maximum possible value.

If a given program point has more than one predecessor instruction (e.g., at the end of *if* structures), it means the variable abstract values must be **joined**. We defined a join operator that merges the possible variable values. For example, if the instruction `a++` has two predecessors that propagated the abstract values 1 and 5 for variable *a*, then the join operator transforms those values in a single one: $[1, 5]$. This means that variable *a* can have two different values at that entry point, which is in fact true. The propagated result after the instruction `a++` will then be $[2, 6]$.

Once the analysis reaches a loop, it knows all the possible abstract values for those variables, and tries to symbolically execute such loop with them. For every iteration of the loop the stop condition is verified, and if it fails for **all** possible values of the variables, then we have reached the upper bound of that loop.

To ensure that this analysis will finish, it is necessary to provide an exagerated upper bound, for the case that the loop condition never fails for the possible abstract values (when the map structure *M* does not have an entry to every variable with input values assigned to it). This value can be parameterized, according to the program under analysis. For example, for a program calculating the length of a word, this value should be the maximum length of any word under consideration.

3.3 Combining SPL Static Analysis With Energy Estimation

In subsection 3.1, we explained how to predict the behavior of hardware components in a feature-oriented manner. This analysis allows us to predict how the hardware will behave after executing every statement, and for every valid *product* in the SPL.

Similarly to SPL static analysis, the result will be stored in a CFG with a *lifted lattice* $\mathcal{L}$ in each node, representing the hardware components states for each *product*, as shown in Figure 6. Here, every $\mathcal{L}_i$ represents a list of *n lattice* elements, one for each of the *n* hardware components considered. For example, $\{A\} \rightarrow \mathcal{L}_1$ (where $\mathcal{L}_1 = \{\mathtt{l}_1, ..., \mathtt{l}_n\}$) means that $\mathcal{L}_1$ holds the states of the *n* hardware components after the execution of `x=input()`, for the product with the configuration $\{A\}$. After executing `x*=2`, the *lifted lattice* $\mathcal{L}_1$ will change to $\mathcal{L}_2$ for products with configuration $\{A\}$ and $\{A, B\}$, since that instruction is only included in *products* with *feature A*.

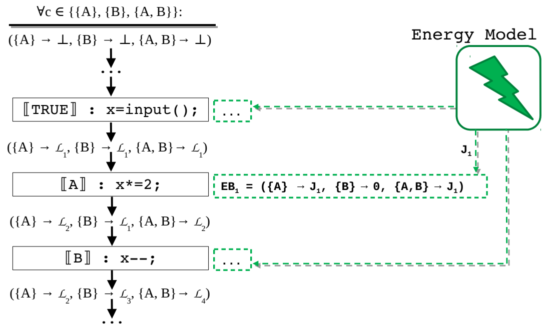

∀c ∈ {{A}, {B}, {A, B}}:
({A} → ⊥, {B} → ⊥, {A, B} → ⊥)
Energy Model
⟦TRUE⟧ : x=input();
...
({A} → L1, {B} → L1, {A, B} → L1)
J1
⟦A⟧ : x*=2;
EB1 = ({A} → J1, {B} → 0, {A,B} → J1)
({A} → L2, {B} → L1, {A, B} → L2)
⟦B⟧ : x--;
...
({A} → L2, {B} → L3, {A, B} → L4)
...

Figure 6: Calculating local energy bounds for each node in the CFG

The information in every node represents only the state of the machine **before** executing the instruction in that node. In order to get an energy estimation, and following the WCET principle, we need to match the states (*lattice* elements) with an *energy model*, $\mathbb{E}$, where the consumption per state is specified. $\mathbb{E}$ can be described as a function that takes as argument a tuple (i, C, s), where i is the node instruction, C is the hardware component, and s is the component state (i.e., the *lattice* element), and returns a consumption value reflecting the work made by the hardware component C to execute i while in the state s. With n hardware components, this function must be invoked n times for each node.

The *energy model* will give a local energy bound for each node in the CFG, which will serve as input to a constraint solving system capable of predicting the energy consumption of the entire program in the worst case scenario based on such local bounds. This will be further explained in subsection 3.4. For the purpose of validating our work, we created a specific energy model, which we will present in Section 5.1. Our analysis can, however, use different energy models provided by, for example, hardware manufacturers.

After matching all this information with the aforementioned *energy model* the CFG will have, for every node i, a set of local energy bounds EB_i. Each element j of every EB_i will be the local energy bound for a valid *product configuration j*. If the instruction in the node is not included in the *product configuration j*, then its local bound will be 0.

Figure 6 represents part of the CFG of Example 1, where it is possible to see the *energy model* assigning an energy value to all *product configurations* in a node. In the example, since the instruction is included only in products $\{A\}$ and $\{A, B\}$, they both receive the value J_1, while the *product* $\{B\}$ gets 0. Also, the value J_1 reflects the combined consumption of all hardware components, and so it depends on the list of *lattice* elements represented by $\mathcal{L}_1$.

3.4 Worst Case Prediction

The primary goal of this work, as we said before, is to determine the energy consumed by every *product* in a worst case execution scenario. Until now, we have shown how to determine the energy consumption of each statement individually, concerning the context in which it executes. However, we need to estimate the overall consumption of a *product*, considering that not all statements have the same impact, for example loop statements will probably execute more frequently, and *if* statements may not always execute.

There is a widely used approach in WCET prediction called IPET - *Implicit Path Enumeration Technique*[12]. This technique consists of translating the CFG into a system of constraints. Such constraints are a result of combining program flow and CFG node execution time bounds, with the program WCET estimation being the result.

From static analysis, each node i has a local execution time upper bound, t_i, expressing the contribution of the node's statement to the total execution time, when executed once. IPET considers a new variable, x_e, where e is an *entity* which can be a node or an edge between two nodes. This variable represents the number of times e is expected to execute, and all these count variables are subject to constraints reflecting the program's structure and possible flows.

The constraints for IPET must follow a basic but strong principle: the number of times an entity e is accessed, x_e, **must** be equal to the sum of the number of times each of its predecessors are accessed, while also being equal to the sum of the number of times each of its successors are accessed. In other words, for every entity e,

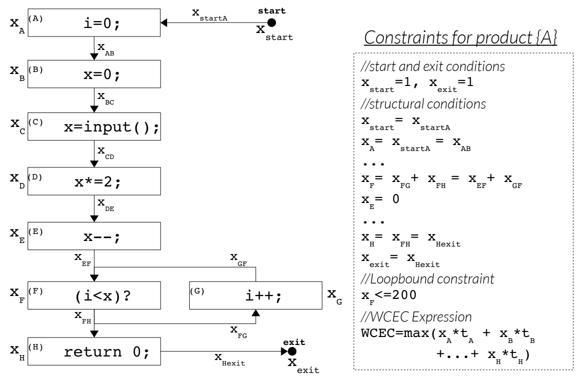
$$x_e = \sum x_i = \sum x_j$$

for all entities i and j that are predecessors and successors of e.

For our purpose, the local upper bound for each node (t_i) will be the one determined by using the method described in subsection 3.3, i.e. the energy consumption local bound. IPET will take care of determining the values for every x_i, in a way in which it maximizes the overall energy consumption. In other words, the worst case energy consumption *WCEC* (just as WCET) is calculated by maximizing a function defined as the sum of all t_i multiplied by its corresponding x_i. In other words, with N instructions in a program, the WCEC function for a *product* is defined as:

$$WCEC = max(\sum_{i=0}^{N} x_i * t_i)$$

Figure 7 shows how the constraints for the product $\{A\}$ of Example 1 would look like. Note that the loopbound constraint indicates that the loop will execute at most 200 times, since the loop is upper bounded by the value of the variable x, and it's maximum value at that point is 200. For product $\{B\}$, this constraint would be $x_F \leq 99$.



```
//start and exit conditions
x_start=1, x_exit=1
//structural conditions
x_start= x_startA
x_A= x_startA = x_AB
...
x_F= x_FG+ x_FH = x_EF+ x_GF
x_E= 0
...
x_H= x_FH= x_Hexit
x_exit= x_Hexit
//Loopbound constraint
x_F<=200
//WCEC Expression
WCEC=max(x_A*t_A + x_B*t_B
              +...+ x_H*t_H)
```

Figure 7: IPET constraints for product $\{A\}$ of Example 1

The IPET technique could already be used to determine the worst case energy consumption scenario of a *product*. However, we have to provide information about statements to be included/excluded in the *product*, since we do not want to consider the energy consumption influence of statements from excluded features. For this, we can set a few extra restrictions that will assure the exclusion of feature-dependent statements not included in the *product* being analyzed. This will be a restriction such as $x_e = 0$, for all entities e which are instructions and not part of the product being analyzed.

Using this strategy we may then calculate the worst case energy consumption scenario for each *product* in a SPL. Considering the 4 *products* included in Example 1 (with configurations $\{A\}$, $\{B\}$ and $\{A, B\}$), our technique determines the *product* with configuration $\{B\}$ is the most energy efficient. This is due to the fact that the loop for this *product* is bounded to 99 iterations, roughly 50% less than the loop bound for $\{A\}$ (200 iterations) and $\{A, B\}$ (199 iterations). This causes the amount of work to be substantially less in $\{B\}$, thus the energy consumption will also be lower.

4 THE SERAPIS TOOL

To validate our technique, we developed a prototype tool, named *Serapis*. This tool is able to, given the source code of a SPL, statically reason about the energy consumption of its *products*.

We implemented our technique using 3 distinct languages: both C and Python for the energy model generation and for the dynamic energy measuring framework, and Haskell for the actual WCEC prediction tool in SPLs. In this section, we will explain the workflow of the tool, as well as its capabilities and limitations.

The workflow of *Serapis* is divided in 3 steps. The first step consists of creating the feature-sensitive CFG for the SPL, following the principles presented in section 2. Our approach in this step was to create the CFG from the Abstract Syntax Tree (AST) representing the analyzed SPL. In order to simplify the source code analysis, we considered a simplified intermediate representation of the C language, called **C Intermediate Language - CIL** [1]. This language works as a subset of the C language, making it possible to have all the functionalities allowed by the C language but reducing the syntactic sugar to the minimum (for example, *for* loops and variable initialization at declaration time are not allowed).

Using the tools provided by CIL, we are able to transform the source code in the SPL from C to CIL, parse the code and create the AST representing the program. The CFG was created using an Haskell library for graph creation and manipulation [2]. As explained in **??**, every node in the CFG contains information about a source code statement and the *features* that implement it.

The pre-processor references to *features* (as explained in subsection 2.1) also need to be included in order to identify code blocks belonging to specific *features*. Since such references were not parsable by CIL, we developed a domain specific pre-processor capable of transforming such references into parsable statements. Our pre-processor transforms every `#ifdef <feature>` primitive in a function call to the form `__feature_<feature>()`, and every `#endif` in a function call to the form `__endFeature()`. At the CFG construction phase, we used those function calls to determine the *features* which implement each statement. Every time a function call appears in the first form, the statements which follow are assigned to that *feature*, until the corresponding `__endFeature()` call appears.

The second step is the static analysis step, which corresponds to the first three phases of the strategy presented in Section 3. Here, the CFG is given as input to a series of functions, each one responsible for a specific type of analysis:

- **Energy behavior analysis**: determines the machine state of hardware components after each statement execution, using the fixed-point computation technique.
- **Flow analysis**: determines the variable types and values before each loop execution, and symbolically executes each loop to determine its upper bound.
- **Local energy bounds**: considering the result of the two previous analysis, the energy model calculates a local energy bound value for each *product*, on each node.

Once all the analysis are finished, the result is passed to the third and final step: the overall energy prediction. In this step, the CFG and the loop upper bounds are used to create the constraints and the maximization function needed by IPET (as described in subsection 3.4). For the maximization function, the cost of each statement is given by the energy model, along with the information from the other analysis. To implement this, we used an integer linear programming (ILP) library for Haskell, which allows to create the maximization function along with a set of constraints, and has an implemented algorithm to compute the solution for such a problem [3]. The overall workflow of *Serapis* is described in Figure 8.

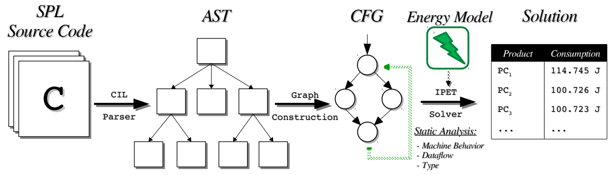


Figure 8: The *Serapis* Workflow

Our tool shares all the advantages of the monotone framework for static analysis [14], thus the correctness properties are maintained. Its accuracy can only be increased by improving both the energy and machine behavior models. Nevertheless, it also has the same limitations. The real unavoidable limitation of static analysis is the fact that recursive functions cannot be considered. Given that, we used a SPL that has no recursive functions to evaluate it.

Moreover, the dataflow analysis process, used to determine the loop upper bounds, is very costly in terms of execution time and memory usage. For an example with around 500 loops and 1500 variables, our tool gave accurate upper bounds, but it did not end in an acceptable time (took several hours to finish), so it was not convenient to use this approach while testing and refining the tool. Given that, we inspected all the loops in the analyzed SPL to see what was the worst case scenario in terms of number of executions, and we manually assigned that value to each loop. However, we have designed some improvements to implement in the dataflow analysis step, in order to significatively reduce the time it takes to obtain the upper bounds. Such improvements include memoization and exclusion of variables not used in the loop conditions.

At the moment, *Serapis*[4] has around 2330 lines of code, divided in 11 modules. The model generator tool and the dynamic energy measuring framework have 500 and 170 lines of code, respectively.

5 EVALUATION

This section describes the experiments that we designed and conducted to evaluate the technique that we have proposed in section 3 to statically predict the energy consumption of *products* in a SPL.

In our experiments, we analyzed 7 different *products* within a SPL taken from the disparity benchmark of the San Diego Vision Benchmark Suite [24]. For our analysis, we implemented a framework for dynamically measuring the energy consumption of a *product*, in addition to the tool implementation that we described in section 4. Then, we assess our technique by comparing the observed consumptions against our static predictions.

As we described in subsection 3.3, our technique needs an energy model with energy consumption values predicted for single instructions. So, in subsection 5.1 we start by describing our approach to generate an energy model which can be used by our technique.

All studies were conducted on a desktop with the following specifications: Linux Ubuntu 14.04 LTS operating system, kernel version 4.4.0-59-generic, with 6GB of RAM, a Ivy Brigde Intel(R) Core(TM) i5-3210 CPU@2.50GHz

[1]More information can be found here: https://people.eecs.berkeley.edu/~necula/cil/.
[2]The functional graph libray (fgl): https://hackage.haskell.org/package/fgl.
[3]The ILP package: https://hackage.haskell.org/package/hmatrix-glpk-0.5.0.0.
[4]For reviewing purposes, *Serapis* is available at https://bitbucket.org/marco10/serapis.

5.1 Static Model Creation

Both the energy and prediction models are crucial components needed by our technique. The results' accuracy depends on the energy values' precision in the energy model, and on the accuracy in detecting behavior changes in the hardware.

These kind of models are allways hardware/language dependent, which means that we need to follow a modelling approach previously presented and build our own models. In fact, modeling the hardware behavior requires exhaustive knowledge about it: how it was programmed, what are the possible execution states, what is necessary to transition to/from a state, etc. Since there is not, to the best of our knowledge, an adequate study showing the influence of hardware behavior on energy consumption, for now we were only able to create a machine with a single state per hardware component. Given this, the consumption of each instruction on that state will be an average of several measurements. This will result in a *lattice* with only one element, and therefore the *transfer function* is the identity function, since the *lattice* element will not change.

In order to get the energy measures, we developed a dynamic framework capable of executing single instructions and obtain energy estimations for these. We used the Running Average Power Limit (RAPL) tool, which provides accurate energy measurements [6]. For now, we are only exploring RAPL's ability to provide CPU-related energy consumption estimations. Thus, we created an energy model only for the CPU hardware component.

The energy modeling process of single instructions varies between instruction types. Following the approach for energy modeling presented in [7], source code instructions may basically be of two types: (1) single-cost operations or (2) API/function invocations. Single-cost operations are source code instructions where energy cost is constant, and it depends only on the number of operands it has. Arithmetic and logic operations, such as a division or a xorl, are examples of such instructions. The cost of such an operation is composed of a constant consumption $\mathbb{C}$, plus an offset $\mathbb{O}$ which is directly proportional to the number of operands.

Estimating the energy cost of an API/function invocation requires a different technique. In most cases, we do not have access to the source code of an API/function, so we must have an estimation for its energy cost, which may depend on several things. First, it inevitably depends on the number of arguments of the function, as they need to be pushed to the stack before the function call. Second, it can depend on the value of the arguments, since there might be a loop in the function parameterized by the arguments, or it may have a premature return instruction included, which can be triggered depending on one or more arguments. The type of the arguments may also play a role in the energy cost (ex. reading and storing a double is different than reading and storing an integer.)

In order to address these problems, we had to look at each one individually. For functions with a variable number of arguments or types (such as `scanf`), we tested them several times, each time with an increasing number of arguments, or with different types. After obtaining the consumption values, the framework tries to detect if there is a relation between the number of arguments (of the same type) and the energy consumed, i.e., if diving the consumption for the number of arguments results in an equal value $\mathbb{C}$, or in values with a slight difference between them. If so, the consumption value for that function will then be the constant value $\mathbb{C}$ multiplied by the arguments length. The same principle was applied when the arguments types were variable, except that the consumption value was divided by the type size in *bytes* (ex. integers need 4 *bytes* to be stored, while chars need only one).

The framework workflow for estimating the energy cost of both types of instructions is identical. For a given instruction `i`, the framework executes it 20 million times. This ensures a measurement of a sufficiently long duration that exceeds the sampling interval of our energy measurement tool (RAPL). We repeated this process 200 times, to reduce the impact of cold starts, cache effects or background processes, and we obtain the energy consumption for each time. Using these values, we calculate the average value, after removing the outliers (5 highest and lowest values).

In most cases, single-cost instructions (or even functions) depend on other instructions. For example, the instruction `a=b*2` is an assignment to the variable `a` of a multiplication between the variable `b` and the constant value 2. In order for this to work, the variable `b` needs to be already declared and have some value assigned to it, but we only want to know the cost of `a=b*2`. To address this problem, we have assigned, to each instruction/function, the list of independent instructions they depend on.

Given the average value for each instrucion/function `i`, calculated after the 200 measurements, we subtract the average value of each of the dependencies. The final consumption will be the resulting value. Figure 9 shows the overall process for energy modeling of both single instructions and functions. For our evaluation purpose, we modeled the energy consumption for 18 functions from the C library, which is the set of functions used in the disparity benchmark referred before.

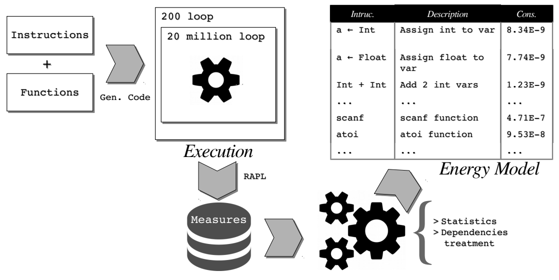


Figure 9: Workflow of the Energy Model Creator

5.2 Experimental Evaluation - Methodology

In order to evaluate WCEC and *Serapis*, we have used them to analyze the energy consumption of a real SPL.

Indeed, we have taken seven *products* within a line obtained from the disparity benchmark, from the San Diego Vision Benchmark Suite [24]. We are using the conversion to the C code given by [19].[5]

The disparity benchmark is able of calculating the disparity between two images, which can be used to detect the depth of objects in an image. We used this SPL for the evaluation process since it is a real-world software system, widely used for research purposes, and also because it avoids the static analysis limitations.

All seven *products* from the SPL share most of the source code, but differ on its implementation for computing the disparity between

[5]Although there are 2 more *products* in the line, they do not share any code with the other 7, so we have discarded them here.

2 images: given a pair of images for a scene, taken from slightly different positions, the disparity map algorithm computes the depth information for objects jointly represented in the two pictures [6].

The so called disparity map algorithm is the variability component, and it gives rise to 7 different and exclusive *features*, which are: *Original_noZ3*, *Original_Z3*, *Unchecked*, *Modified1*, *Modified2*, *No_Intermediates* and *Manual_Interchange*. In other words, it is mandatory for a *product* to include only one of this *features*.

WCEC Accuracy: The first step of the evaluation process consisted of determining the actual worst-case energy consumption value for each *product*. In order to do this, we used the *brute-force* algorithm: we generated all the 7 *products*, instrumented the source code with energy measurement calls, and then executed each *product* 200 times. This was performed by our dynamic energy measuring framework, which is responsible for executing the *products* with the same input and measure the energy consumption. Since our goal is to determine the energy consumed in the worst case, we retrieved the highest value for every *product* in the 200 measurements obtained [7]. Just like for the energy model creation, we used RAPL to get the actual measurements, which minimizes the instrumentation overhead by simply measuring the energy before and after the program execution.

By combining the created energy model, as described in subsection 5.1, with our technique (implemented in a tool, as described in section 4), we were able to compute accurate estimates for the worst case energy consumption of every *product*. The results obtained are presented in Table 1.

Table 1: Study Results

Product	LOC	Observed WCEC	Predicted WCEC	Diff.	% Error	Exec. Time	Pred. Time
{Original_noZ3}	2938	114,17 J	114,75 J	0,58 J	0,50%	13.87s	409.57s
{Original_Z3}	2550	90,80 J	100,73 J	9,93 J	10,93%	11.57s	161.48s
{Unchecked}	2503	87,93 J	100,72 J	12,79 J	14,54%	10.97s	165.69s
{Modified1}	1884	64,91 J	66,54 J	1,63 J	2,52%	8.54s	35.98s
{Modified2}	1801	64,47 J	70,08 J	5,62 J	8,71%	8.58s	29.08s
{No_Intermediates}	1753	58,87 J	64,93 J	6,07 J	10,31%	8.01s	13.24s
{Manual_Interchange}	1754	56,80 J	67,06 J	10,26 J	18,07%	7.40s	13.66s

As we can see, the consumption values which the technique predicted were always higher then the measured ones, which is adequate since we want to predict the worst case. Moreover, it has a maximum error percentage of 18.07% and a minimum of 0.5%, while the maximum difference in joules is only 12.79. These are very promising values, since the error percentages and difference values are considerably low, considering that this is the first approach for energy consumption prediction in SPLs.

WCEC Performance: Our static analysis technique works at the source code level. Thus, it can be seen as an interpreter which, at execution time, parses the code, builds data structures (ASTs and CFGs), and performs complex computations. The WCEC analysis of a specific *product* will always take more time than executing its compiled (and optimized) code, as the results of Table 1 confirm (see results displayed in the last two columns). However, it should be noticed that, to measure the energy consumption a of *product* in a SPL, developers have to generate the *product*, instrument it with energy monitoring code, and finally run the *product*. This overhead is not reflected in the execution times shown in Table 1.

[6]The *products's* source code is avaiable at http://specs.fe.up.pt/publications/array16.zip.
[7]The 200 measurements have similar consumptions (with small standard deviation).

Regarding static analysis, the technique that we propose assures that the source code of every *feature* is analyzed only once, contrary to the use of traditional static analysis techniques. The exception is the worst case prediction technique shown in subsection 3.4, which works at the *product* level. This means the solver will run once per *product* and thus repeat the calculations for shared features. In the future we intend to improve the performance of *Serapis* by encoding a SPL in the solver in a different way so it does not recalculate the constraints for shared features.

6 RELATED WORK

Energy consumption awareness has brought up an increasing interest in analyzing the energy efficiency of software systems. Developers seem to be now more focused on reducing energy consumption through software improvement [18], since it is the software that triggers the hardware behavior. This principle guided several research works that appeared in the last decade.

Studies have shown that the energy consumption of a software system can be significantly influenced by a lot of factors, such as different design patterns [16], data structures [17], and refactorings [20]. Even in software testing the decisions made influence the consumption at the testing phase [11]. In the contex of mobilie devices, there are other works that focus on analizing energy per software application [9], or even compare different usages of similar applications [8], while others tried to determine the energy consumed by code blocks, such as functions/methods [13] or lines of code [10]. Most of the works are based on energy models: a prediction model able to determine the energy consumption by matching information retrieved from hardware components to previously obtained measurements [15, 26].

More recently, a few more works appeared which were focused not in measuring energy, but in predicting the energy consumption of different programs [3]. This was achieved by combining program analysis tools and techniques, and more detailed energy consumption models (instruction oriented instead of hardware oriented). The result of these studies is an estimate of the energy consumed by a certain program in a specific scenario.

Regarding software product lines, Thüm et al. survey analysis strategies [23], but they do not explore data-flow analysis approaches, neither performance or energy consumption estimation. Related work on data flow analysis [1, 2] and performance estimation [4, 21, 22] for SPLs share with the surveyed work, and our work, the general goal of checking properties of a SPL with reduced redundancy and efficiency. Similar to the initial phases of our approach, the data flow analysis works and a number of approaches covered by the survey adopt a family-based analysis strategy, manipulating only family artifacts such as code assets and feature model. Contrasting, a fully product-based strategy, such as the generate-and-analyze approach we use as baseline, manipulate products and therefore might be too expensive for product lines having a large number of products. We reduce risks and part of the performance penalty by requiring a per product analysis only in the final phase of our approach. To avoid this kind of deficiency, the mentioned performance estimation work opts for a sampling approach, which is more efficient but does not guarantee the obtained results apply for all products.

7 CONCLUSION AND FUTURE WORK

This paper introduced a technique to statically predict the energy consumed by all *products* in a SPL in the worst-case scenario which we termed *Worst-Case Energy Consumption*. This technique mainly relies on two components: *i)* the combination between the static program analysis concepts and the *Worst-Case Execution Time* technique, and *ii)* the existence of an energy model which assigns energy consumption values to different types of instructions and function calls. We explained how we implemented our technique in a tool called *Serapis*, and also how we created the required energy model by following an already existing approach.

We evaluated our technique by comparing the consumption values predicted for every *product* in a SPL with the actual measured ones. We considered 7 *products* from a SPL consisting of a benchmark for the disparity map algorithm, used for image processing. All 7 *products* had different implementations for that algorithm.

Our results show that it is possible to determine the energy consumed by a *product* without actually executing and measuring it. Indeed, our technique was able to always give accurate estimations, showing a mean error percentage of 9.4% and a standard deviation of 6.2%. The predicted values were also always higher then the measured ones, which was expected since the technique is supposed to predict the worst-case. Nevertheless, the accuracy of our technique can still be improved. The machine behavior model is very simplistic, and thus it does not reflect the real behavior of a realistic CPU. Moreover, improving such a model makes it necessary to adapt the energy model, so it can not only give instruction oriented, but also machine state oriented energy estimations.

Although most of WCEC analysis is feature-oriented, the ILP constraint solver estimates consumption per *product*, not reusing estimations of previously analyzed *features*. Since this is a constraint solver limitation, we plan to explore other techniques for constraint solvers which avoids this restriction. This will improve the performance of *Serapis*, namely in SPLs where *features* are shared between several *products*. The performance will also be improved by using more advanced techniques to compute loop bounds, that we are currently including in our prototype. Finally, we also plan to apply WCEC in optimizing the energy consumption of SPLs.

Acknowledgments

We would like to thank Simão Melo de Sousa (University of Beira Interior) for helpful discussions about the topics of this paper, and to the anonymous reviewers for the valuable comments and feedback. This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-016718, and FLAD/NSF under the project with ref. 278/2016.

REFERENCES

[1] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. 2013. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proc. of 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '13)*. ACM, 355–364.

[2] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. 2012. Intraprocedural Dataflow Analysis for Software Product Lines. In *Proc. of 11th Annual Int. Conf. on Aspect-oriented Software Development (AOSD '12)*. ACM, 13–24.

[3] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, and K. Eder. 2014. Static energy consumption analysis of LLVM IR programs. *CoRR* abs/1405.4565 (2014).

[4] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wąsowski. 2012. Variability-Aware Performance Modeling: A Statistical Learning Approach. (2012), 301–311.

[5] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. 2006. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *2006 27th IEEE Int. Real-Time Systems Symposium (RTSS'06)*. 57–66.

[6] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. 2012. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Perform. Eval. Rev.* 40, 3 (2012), 13–17.

[7] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. 2013. Estimating Mobile Application Energy Consumption using Program Analysis. In *Proc. of 35th Int. Conf. on Software Engineering (ICSE)*.

[8] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann. 2015. EcoDroid: An Approach for Energy-based Ranking of Android Apps. In *Proc. of 4th Int. Workshop on Green and Sustainable Software (GREENS '15)*. IEEE Press, 8–14.

[9] M. Kjærgaard and H. Blunck. 2012. Unsupervised Power Profiling for Mobile Devices. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, Alessandro Puiatti and Tao Gu (Eds.). Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Vol. 104. Springer Berlin Heidelberg, 138–149.

[10] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. 2013. Calculating Source Line Level Energy Information for Android Applications. In *Proc. of 2013 Int. Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, 78–89.

[11] D. Li, Y. Jin, C. Sahin, J. Clause, and W. G. J. Halfond. 2014. Integrated Energy-directed Test Suite Optimization. In *Proc. of 2014 Int. Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, 339–350.

[12] Y. S. Li and S. Malik. 1995. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proc. of 32Nd Annual ACM/IEEE Design Automation Conference (DAC '95)*. ACM, 456–461.

[13] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes. 2016. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In *2016 IEEE 23rd Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 517–528.

[14] A. Møller and M. I. Schwartzbach. 2015. Static Program Analysis. (May 2015). Department of Computer Science, Aarhus University.

[15] S. Nakajima. 2013. Model-based Power Consumption Analysis of Smartphone Applications. In *16th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, September 29th, 2013*.

[16] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva. 2017. Helping Programmers Improve the Energy Efficiency of Source Code. In *Proc. of the 39th International Conference on Soft. Eng. Companion*. ACM. (to appear).

[17] R. Pereira, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva. 2016. The Influence of the Java Collection Framework on Overall Energy Consumption. In *Proc. of 5th Int. Workshop on Green and Sustainable Software (GREENS '16)*. ACM, 15–21.

[18] G. Pinto, F. Castor, and Y. D. Liu. 2014. Mining Questions About Software Energy Consumption. In *Proc. of 11th Working Conf. on Mining Software Repositories (MSR 2014)*. ACM, 22–31.

[19] L. Reis, J. Bispo, and J. M. P. Cardoso. 2016. SSA-based MATLAB-to-C Compilation and Optimization. In *Proc. of 3rd Int. Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2016)*. ACM, New York, USA, 55–62.

[20] C. Sahin, L. Pollock, and J. Clause. 2014. How Do Code Refactorings Affect Energy Usage?. In *Proc. of 8th ACM/IEEE Int. Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. ACM, 36:1–36:10.

[21] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T). In *Proc. of 30th Int. Conf. on Automated Soft. Eng. (ASE '15)*. IEEE Comp. Soc., 342–352.

[22] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. 2015. Performance-influence Models for Highly Configurable Systems. In *Proc. of 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, 284–294.

[23] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45.

[24] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. 2009. SD-VBS: The San Diego Vision Benchmark Suite. In *Proc. of 2009 IEEE Int. Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, 55–64.

[25] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. 2008. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. (2008).

[26] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. 2010. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proc. of Eighth Int. Conf. on Hardware/-Software Codesign and System Synthesis (CODES/ISSS '10)*. ACM, 105–114.