Optimising Energy Consumption of Design Patterns

Adel Noureddine, Ajitha Rajan
School of Informatics
The University of Edinburgh
adel.noureddine@ed.ac.uk, arajan@staffmail.ed.ac.uk

Abstract—**Software design patterns are widely used in software engineering to enhance productivity and maintainability. However, recent empirical studies revealed the high energy overhead in these patterns. Our vision is to automatically detect and transform design patterns during compilation for better energy efficiency without impacting existing coding practices. In this paper, we propose compiler transformations for two design patterns, Observer and Decorator, and perform an initial evaluation of their energy efficiency.**

I. Introduction

Software design patterns [1] are time-tested solutions to recurring design problems, and are widely used by practitioners to provide improved code readability, maintainability and reuse. They also facilitate communication between software engineers. In recent years, energy consumption has emerged as an important design constraint when writing software, especially in the domain of embedded systems where a strict power budget is imposed [**?**].This observation led researchers to evaluate existing design patterns with respect to energy efficiency. Recent studies [3], [4], [5] found that some, not all, design patterns negatively impact energy consumption (up to 712% on embedded hardware). This leads us to the question we explore in this paper, **How can we improve the energy efficiency of design patterns while retaining the essential benefits of improved code readability, maintenance and reuse?** The answer, we believe, lies through compiler optimisations that have the potential for energy savings with *no changes* to existing software or hardware. Existing studies to improve energy efficiency using compilers rely on optimised use of hardware features such as dynamic frequency and voltage scaling. Optimisations targeting software design patterns for energy efficiency is entirely novel. We believe this approach has the potential to change the current state of practice since (1) Developer coding practices remain unaffected, (2) Benefits of using design patterns are retained, (3) Energy consumption of software is reduced. In this paper, we study the Observer and Decorator patterns that consistently performed poorly across all empirical studies, including ours. We propose energy efficient transformations for programs with these design patterns. The program transformations are to be carried out during the compilation stage. As an initial evaluation, we manually transformed 11 programs with the Decorator and Observer patterns and tried to optimise object creations and function calls specific to these patterns. We found our transformations achieved average energy gains of around 10%. Our vision for the future is that given a program with design patterns, we will automatically detect and transform patterns at the compilation stage for improved energy consumption. Our approach addresses the need for energy-aware software while retaining the benefits of design patterns in software engineering.

II. Motivation and Previous Work

Existing studies using compiler optimisations for improving energy consumption propose techniques that optimise applications' usage of hardware features. Approaches include using a mapping algorithm to run different components of an application on different chip cores while independently varying their voltages [6], using dynamic compilers with dynamic voltage scaling (DVS) techniques [7], using compiler-directed DVS for reducing network-on-chip energy [8], or proposing a petri net based model for setting frequencies in multiple clock domain micro-architectures [9].

Our proposed approach of using compiler optimisations to improve energy efficiency of software code design has not been previously explored. However, recent studies investigated the energy benefits of transforming applications based on the execution environment. In [11], a software framework was introduced to transform applications for energy efficiency based on developer's input and energy profiling. The approach requires manual input and creating, running and measuring multiple variations of the application in order to choose one efficient transformation. In contrast, our approach does not require developers' manual input or additional energy measurements. In [10], the authors proposed an approach to rewrite web applications in order to reduce the energy required to display these web pages in a mobile device.

There have been recent studies evaluating energy consumption of design patterns by comparing programs with design patterns against functionally equivalent programs without these patterns. In [3], the authors compared the energy consumption of 6 design patterns on mobile phones. They found a high overhead for the Prototype, Decorator and Factory design patterns (33%, 133%, and 15.9% overhead, respectively). The authors in [4] compared the energy consumption of 3 design patterns. Their results show a negligible energy overhead for the Factory and Adapter patterns, and around 44% energy overhead for the Observer pattern. Finally, Sahin et al. studied the energy consumption of 15 design patterns in [5]. Their results show a high energy overhead for the Factory, Decorator and Observer patterns (712%, 21.55%, and 62%, respectively).

However, they run their experiments on an FPGA board as opposed to an operating system on a regular computer.

The generalisation of these results is limited by the characteristics of the execution environments (*e.g.*, mobile phone, FPGA boards) and the small number of design patterns in the experiments. Although energy is crucial in mobile devices, it is also an important economic factor on desktop computers and servers. To corroborate the results for energy consumption of design patterns, we conducted an empirical evaluation on a modern computer with 21 design patterns, taken from Huston's website [12].

Empirical Study on Energy Usage

The design pattern examples in our study are written in C++ (14 patterns: Mediator, Observer, Strategy, Template, Visitor, Abstract, Builder, Factory, Prototype, Singleton, Bridge, Decorator, Flyweight and Proxy) and in Java (7 patterns: Chain, Command, Interpreter, Iterator, State, Adapter and Composite). We used OpenJDK 1.7.0_65 to compile and run the Java patterns, and Clang 3.5 [13] to compile the C++ patterns. We used JOLINAR 2 [**?**] to estimate the energy consumption of our code at runtime. We ran our experiments on a Lenovo Thinkpad X220, with an Intel Core i5-2540M CPU topping at 2.60GHz. We measured the CPU energy overhead (positive or negative) of a program using a design pattern against a program version without the pattern[1]. We ran each version of each example 100,000 times in a loop in order to get enough execution data for a valid energy estimation, and we repeated each experiment 10 times. The results from our experiments are shown in Figure 1.

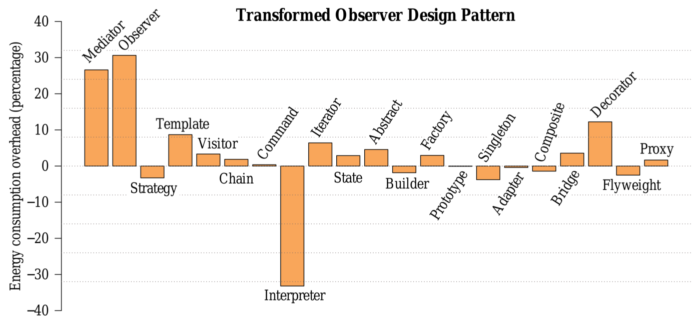

Fig. 1. The energy overhead of design patterns.

As can be seen in Figure 1, the patterns with high energy overhead (>10%) are Observer (30.63%), Decorator (12.24%) and Mediator (26.61%). Our results follow a similar trend to the one presented in [5]. The exact percentages for energy overhead vary slightly since our experiments were run on an actual end-user computer rather than a specific FPGA board.

Having ascertained the energy usage of design patterns, the next step was to select and optimise design patterns that consistently consumed high energy. We picked the *Decorator* and *Observer* patterns since they have a high overhead (greater than 10% across all studies), and are widely used. We did not pick the *Mediator* pattern because it is less popular, and shares behaviour with the *Observer* pattern (*e.g.*, both aim to reduce coupling).

[1]Memory energy consumption, which according to our model is linear with memory usage was measured but was found to be negligible compared to the CPU energy. Execution time overhead was, in most examples, below 1%, with the exception of Mediator (7.9%) and Decorator (16%) patterns.

III. Reducing the Energy Footprint of Design Patterns

We analyse and optimise the Observer and Decorator patterns using the following steps,

1) We ran experiments to check if existing compiler optimisations reduce the energy overhead of the patterns.
2) Existing optimisations were found to be insufficient in step 1, so we propose a set of transformation rules targeted at improving energy overhead.
3) We evaluate our approach over several small examples (discussed in further detail in the following sections).

For each of the examples in our study we have 3 versions,

***without*:** a version that does not use the design pattern,
***with*:** a version that uses the design pattern, and
***optimised*:** a version where we optimise the *with* design pattern version for energy efficiency.

We iterate the examples 1 million times for each experiment run and repeat the experiment 10 times to mitigate the effect of any hidden bias.

A. Decorator Pattern

The Decorator design pattern is a structural pattern. The goal in this pattern is to *attach additional responsibilities to an object dynamically* [1]. We detail the 3 steps in our analysis and optimisation of the Decorator pattern.

1) Compiler Optimisations: We varied the optimisation flags (*e.g.*, O1, O2 and O3) on the Clang compiler for both versions, *with* and *without* the Decorator pattern. Energy consumption while enabling compiler optimisations drops from 9.82 Joules (using O0 flag in Clang) to 9.61 Joules (with O3 flag) on average for the *without* version, and from 11.38 (O0) to 11.11 (O3) Joules for the version *with* the Decorator pattern. The reduction observed when using compiler optimisations for each version is explained with the number of CPU guest instructions (collected through Valgrind's Lackey tool [15]). These instructions drop from 3 billion to 2.6 billion for the *without* version, and from 5.5 billion to 4.6 billion for the *with* version when compiler optimisations are enabled. Clang's optimisation *-O* flags activate a number of LLVM's analysis and transformation passes on the LLVM intermediate representation (IR) code. These passes perform low-level transformations such as deleting dead loops or dead code, merging basic blocks or combining redundant instructions. Although Clang's optimisations allow energy reductions within each version, we still observe the same overhead of 15.6% (on average) between the *with* and *without* versions. In other words, existing compiler optimisations are not sufficient to improve the energy consumption resulting from the design pattern.

2) Transformation Rules: Our transformations for energy efficiency optimises the number of object creations and function calls in the pattern. These instructions cannot be improved by existing compiler optimisations as such transformation requires specific knowledge of the pattern. The transformation rules for the Decorator pattern are presented below and illustrated in Figure 2:

1) Identify object instantiations using the Decorator classes. For each such instantiation, perform steps 2 and 3
2) Create a sub-class that encapsulates all the attributes and functions of the decorator classes used in this instantiation. For example, we create *SimpleWindowHorizontalVertical* sub-class when we identify a *SimpleWindow* object being decorated with both *HorizontalScrollBar* and *VerticalScrollBar* decorators.
3) Substitute the object instantiation using decorated classes with instructions to create an object of the new sub-class.

The proposed transformation replaces multiple object constructions (an object and its decorators) with a single object creation (an object from the added sub-class). It is worth noting that other transformations of the Decorator pattern for energy efficiency may exist. We, however, have not investigated other transformations since the goal in this paper is to show that a transformation of design pattern for energy optimisation is possible rather than comparing and presenting the best possible transformation.

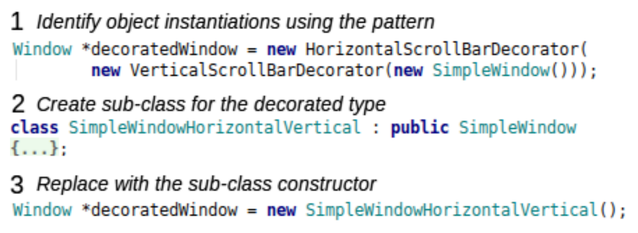
1 *Identify object instantiations using the pattern*
```
Window *decoratedWindow = new HorizontalScrollBarDecorator(
          new VerticalScrollBarDecorator(new SimpleWindow()));
```
2 *Create sub-class for the decorated type*
```
class SimpleWindowHorizontalVertical : public SimpleWindow
{...};
```
3 *Replace with the sub-class constructor*
```
Window *decoratedWindow = new SimpleWindowHorizontalVertical();
```

Fig. 2. An example of Decorator pattern transformation.

We plan to encode the proposed transformation as a compiler optimisation (into existing compilers such as Clang [13]) in the future. In the next section, we evaluate the energy gains from our transformation.

3) Empirical Evaluation: We used 6 programs with the Decorator design pattern from GitHub with sizes ranging from 77 to 198 SLOC. We manually transformed each of these programs using our rules in the previous Section III-A2. The results, in Figure 3, show a marked improvement in the energy overhead after applying our transformation. On average, we achieve a 12.23% reduction in the energy overhead with our transformations, with improvements ranging from 4.71% up to 25.47%. The difference in percentage improvements across our examples is due to the extent to which the pattern is used in them. For example, the *Pizza* program decorates objects with one decorator while the *Sandwich* program has multiple decorators associated with an object. As a result, our transformation removes more object creations in the *Sandwich* program, and thus a larger percentage energy gain, than in the *Pizza* program.

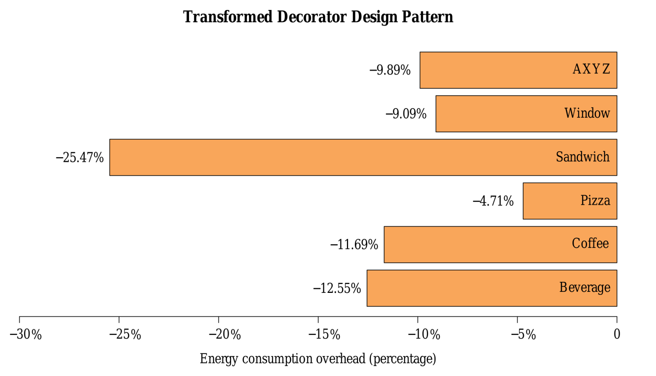

Fig. 3. The energy impact of the transformation rules on the Decorator design pattern.

Functional correctness: In order to validate that our transformations did not modify the functionality of the programs, we compared the outputs of both versions, *with* and *optimised* for all 6 programs using a diff utility. We found the outputs from both versions were a perfect match in all our experiments.

In the next section, we study the Observer pattern, and propose a set of transformation rules in order to reduce its energy footprint.

B. Observer Pattern

The Observer design pattern is a behavioural pattern. The goal in this pattern is to maintain a *one-to-many relation between subject and objects* [1], allowing *observer* objects to be notified and updated automatically whenever the subject changes its state. We detail the 3 steps in our analysis and optimisation of this pattern.

1) Compiler Optimisations: We apply Clang's optimisations to both the *with* Observer pattern and *without* versions. Energy consumption drops from 9.85 to 9.36 Joules on average for the *without* version, and from 13.25 to 10.17 Joules for the *with* version due to the reduction in the number of CPU guest instructions. In contrast to the Decorator pattern, Clang's optimizations were effective in reducing the energy overhead between the *with* and *without* versions, from 34.5% down to 8.64%, on average. Nevertheless, a considerable energy overhead remains and we attempt to reduce this in our transformation of the Observer pattern.

2) Transformation Rules: In this pattern, every change in the subject's state is notified to all observers who then execute an update function. It is usually the case that each of those update functions will constitute a *get* of the updated subject's state. This *get* operation is typically repeated across updates in all the observers. We also inline subscribe/unsubscribe function calls.

Our transformation of the Observer pattern optimises the repeated *get* operation by querying the subject state once and reusing it in all the observer updates. We hypothesize that transforming multiple function calls and memory accesses into a single call will allow a reduction in the energy consumption.

Note that, unlike the Decorator pattern which is structural and creates multiple objects, there is no opportunity to optimise object instantiations in the Observer pattern since it focuses on the communication between objects.

As before, these transformations are to be applied during compilation, so as to retain the advantages of the design pattern for the developer. We applied these transformations manually on the source code of the examples in the next Section and plan to encode them into a compiler in the future.

3) Empirical Evaluation: We transform 5 different programs using the Observer design pattern and report the energy consumption overhead when applying our transformation rules. The programs are taken from GitHub repositories and their sizes range from 80 to 123 SLOC. The results, in Figure 4, show varying improvements in the energy overhead for the different examples. On average, we achieve a 6.95% reduction in the energy overhead with our transformations, with improvements ranging from 4.32% up to 13.02%. As with the Decorator pattern, the percentage improvements vary based on the extent to which the pattern is used in the different examples (*e.g.*, number of observers and number of update notifications).

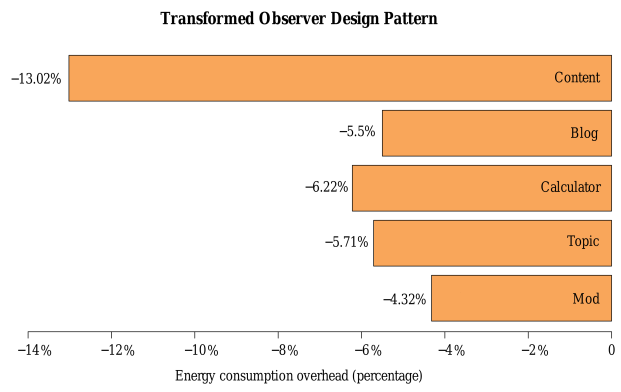

Fig. 4. Energy impact of the transformation rules on the Observer design pattern.

Functional correctness: As with the Decorator pattern experiments, we compare outputs before and after applying the transformations. For all experimental runs of all 5 programs, the outputs from both the *with* and *optimised* versions were a perfect match. These results serve as initial evidence that our transformations do not change the functionality of the code using the design pattern.

IV. Conclusion and Future Directions

In this paper, we present an approach to improve the energy efficiency of software by optimising design patterns automatically at compile time. We explored simple transformations for the *Observer* and *Decorator* patterns and found energy reductions in the range of 4.32% to 25.47%.

Our vision for achieving energy efficient software is to have compiler optimisations that detect and transform design patterns for improved energy consumption. We will leverage existing research for automated detection of design patterns [**?**], [16], [20], [19], [17]. We, then, plan to build compiler transformations in compiler frameworks such as LLVM/Clang. We will establish functional correctness of the compiler transformations and perform empirical evaluations investigating energy gains on industry sized software. The proposed approach not only addresses the need for energy efficient software, but also ensures that current coding practices remain unaltered, a feature desirable for industry adoption.

graphystyleunsrt