The Effect of Automatic Stubbing on Measurement of Energy Consumption

Master's Thesis

J.M. Robeer

Utrecht University
Department of Information and Computing Sciences

July 20, 2018

Author
J.M. Robeer
Utrecht University
Student ID: 3802337

First Supervisor
dr. J. Hage
Utrecht University

Second Supervisor
dr. ir. J.M.E.M. van der Werf
Utrecht University

First External Supervisor
ir. R. van Vliet
Centric

Second External Supervisor
dr. E.A. Jagroep
Centric, Utrecht University

Abstract

Focus on sustainable software has been increasing the past few years. There is a huge potential in the optimization of software applications for energy efficiency. Insight in the energy consumption of software applications is needed for software producing organizations to optimize their applications for energy efficiency. In this thesis we propose an extensions to the Stubbed Energy Profiling (StEP) method of Jagroep *et al.* by automating the creation of stubs using the capture and replay technique. A tool is developed which implements the automatic creation of stubs. The tool and extended method are used in a preliminary experiment and case study to show the feasibility of the approach. The extended method reduces the manual effort required to apply the StEP method, and enables the measurement of energy consumption of software applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the rise of energy consumption, sustainability is gaining more and more attention. Growth of renewable energy resources such as solar power and wind power, is surpassing the growth of traditional energy resources such as fossil and nuclear energy over the past decade [2]. Because Information Technology (IT) solutions are ubiquitous, sustainability has attracted the attention of researchers. Research focuses on creating a more sustainable environment with the help of IT solutions, and by optimizing the energy efficiency of IT solutions. There is a huge potential for the optimization of energy efficiency of IT solutions, because software can account for up to 66 % of the energy consumption of systems [3].

Following up on the work of Jagroep [4], we will consider in detail how to automate energy consumption measurements of realistic software applications under development at the case company Centric. The goal is to assist developers in measuring the energy consumption of software components within applications during the development process, to optimize the energy efficiency of the applications. In this thesis, we continue on the work "The Hunt for the Guzzler : Architecture-based Energy Profiling using Stubs" by Jagroep *et al.* [1]. The StEP (Stubbed Energy Profiling) method proposed in their paper is extended such that it can be applied in an automated fashion.

In Chapter 2 we give an overview of the current state of the research on green software, and introduce concepts used by the method proposed in this thesis. Chapter 3 outlines the research approach used. We formulate the research questions which we will attempt to answer in this research. Chapter 4 presents the method developed to automate the creation of stubs, and discusses the tool which implements the method. Chapter 5 describes the extensions to the StEP method, illustrating how software energy consumption measurements should be performed when using the method proposed to automate the creation of stubs. In Chapters 6 and 7 the proposed method is tested using a set of experiments and a case study to determine the feasibility of applying the method. Chapter 8 concludes this thesis by summarizing the presented work, answering the research questions, and outlining opportunities for future research.

Chapter 2

Background

In this chapter we discuss the background presented in literature on the main topics used throughout the Master's Thesis. First, in Section 2.1 we discuss the background of the field of Green IT. Second, in Section 2.2 we discuss the background of the C# programming language, which is used for the method proposed in the following chapters.

2.1. Green IT

The field of Green IT proposes a holistic approach to follow toward creating a more sustaining environment [5]. The holistic approach encompasses four paths to follow: usage, disposal, design, and manufacturing of IT systems in a sustainable manner [5]. Green IT can be divided into greening *by* IT and greening *of* IT [5]. Greening *by* IT is applying IT systems, within IT and other fields, to create a more sustainable environment. Greening *of* IT is optimizing IT systems for sustainability to create a more sustainable environment.

There has been a focus on the greening of hardware for years, the focus on greening software is behind that of greening of hardware [6]. Major gains are still to be made in greening software, as software remains the main contributor of consumed energy in IT [5, 7]. For example, software can account for up to 66 % of the energy consumption of systems [3], which demonstrates the potential for the greening of software.

2.1.1. Green Software

Green software is defined as "software, whose direct and indirect negative impacts on economy, society, human beings, and environment that result from development, deployment, and usage of the software are minimal and/or which has a positive effect on sustainable development" [8]. The definition of green software encompasses the four dimensions of sustainability: economic, social, environmental, and technical [6]. Greening of software focusses on the environmental dimension of sustainability by optimizing software to reduce energy consumption. Thereby, reducing the carbon dioxide emission that can be attributed to the software, and resulting in a more sustainable environment.

The focus on the environmental dimension also impacts the three other dimensions. To give an example. To perform the software optimizations, the developers of the software choose for some set of libraries to use in their software which are known for their low energy consumption. The choice for the set of libraries affect the long-term use of the software, as the software depends on the (active) development of the libraries (technical dimension). Reducing the energy consumption of the software contributes to an decrease of the cost of software (economic dimension), which results in capital which can be invested in new employees (social dimension).

A lot of effort has been put in reducing the energy consumption of hardware, however software is the true consumer of energy in systems [9] data centers running cloud applications on thousands of servers form the backbone of the fast-growing IT industry [10]. Greening these data centers provide a huge potential for energy savings [10, 11]. Deploying green software on data centers benefits the progress towards the sustainable goal of Green IT [9].

2.1.2. Measuring Software Energy Consumption

Accurate energy consumption measurements are required to create energy-aware optimizations when developing green software [12]. Measuring software energy consumption can be grouped into three categories: hardware measurements, power models, and software measurements [13].

Hardware measurements offer high-precision measurements, but at the cost of course-grained measurements as they only measure the energy consumption of the system as a whole. Power models estimate the energy consumption based on mathematical models. The models are often too generic, or dependent on a specific platform [13]. Software measurements — also known as energy profilers (EPs) — use statistical sampling or code instrumentation. They decompose energy consumption for each resource utilized by the software. For example, using CPU utilization and Random Access Memory (RAM) usage. Energy profilers often include power models to compute the energy consumption estimations [13]. The resource utilization of software measured by the EP is used as input for the power model.

Of the three categories, according to Noureddine *et al.* [13], software measurements are the most promising approach. The works of Jagroep *et al.* [1] can be categorized under software measurements. Hardware measurements are used by Jagroep *et al.* to validate and calibrate the software measurements.

Various energy profilers have been reviewed and compared in recent research [13, 14]. Although energy profilers still have their shortcomings [14], this can be overcome by also including hardware measurements. Jagroep *et al.* found significant differences in the energy measurement of energy profilers compared to the actual energy consumption [14]. By also including hardware measurements, the measured difference in energy consumption can be corrected for.

2.1.3. Software Architecture

Software architecture in relation to Green IT can be divided into three areas of contribution: quality attributes, software architecture perspectives, and tactics. The contributions follow the software architecture guidelines of Rozanski and Woods as presented in [15].

Quality Attributes

To position sustainability within software architectures, existing literature proposes sustainability as a quality attribute (QA) [6, 16]. The proposals in the literature follow the format of ISO 25010, the standardized quality model of software products [17]. A quality attribute is a "measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders" [18]. By including sustainability as a quality attribute in the software architecture, trade-offs can be made during the software development process to satisfy the target sustainability of a software product.

Software energy consumption can be positioned as part of the sustainability quality attribute, along with its measures and measure elements. The measures include software utilization measurements, energy usage, and workload energy of tasks. This allows software energy consumption to be quantitatively evaluated [16].

Software Architecture Perspectives

The software energy consumption perspective [16] provides a means to identify, measure and analyze the architectural elements behind energy consumption. The perspective enables stakeholders to consider sustainability as part of the design of a software product. To apply the software energy consumption perspective a set of activities is provided by the authors [16]. Creating an energy profile of the software product is one of the activities. The energy profile relates the software architectural elements to their energy consumption. It is constructed by performing software energy consumption and performance measurements.

Tactics

To address the concerns of the stakeholders in the software architecture various tactics are proposed [16, 19]. The tactics provide guidelines how the desired effect of sustainability as a quality attribute can be attained. The proposed set of tactics is not yet complete, and are subject to further research [16].

2.1.4. Software Testing

Automating software testing has been extensively researched for the past four decades [20]. Research has focused on: the test oracle problem [20], the generation of input for test cases [21], the extraction of software interfaces [22], and the automatic generation of unit tests [23]. In many areas of software testing research human input is often still required to determine whether the observed behavior in tests is correct.

The automatic generation of unit tests is performed for test factoring to increase test efficiency, or for exploring behavior not exercised by existing unit tests [23]. Test efficiency is creating and running tests more efficiently [23]. Test factoring creates unit tests from system tests, where each unit test exercises a subset of the behavior of the system test [24]. Exploring behavior not exercised by existing unit tests can be done by modifying the state of objects to explore new paths (OCAT [25]), or by mining dynamic traces of program executions (DyGen [26]).

Test factoring creates unit tests automatically by applying the capture and replay method. The created unit tests contain stubs which simulate the environment of the software element under test. A stubbed software element replaces some software element by simulating its functionality [27]. Capture and replay does not modify the original behavior of the program under test.

Capture and Replay

Capture and replay is a technique to improve test efficiency, traditionally used for graphical user interface (GUI) or web application testing, by creating and running test inputs more efficiently [23]. For example, the technique can be used to automatically create unit tests from the execution of a regression test or a system test.

The technique consists of two phases. First, the *capture* phase monitors the interactions between the element(s) under test and its environment during execution of the software system. Based on the monitored interactions, unit tests are generated for the element(s) under test. Each unit test uses the monitored interactions of the element under test as test input, and provides test oracles based on the return values of the elements. Second, the *replay* phase reruns the created unit tests.

Various researchers applied the capture and replay technique to software testing. Saff *et al.* [24] applied the capture and replay technique to automatically create unit tests from system tests for Java applications, showing the feasibility of using the capture and replay method to automate test creation. Orso *et al.* [28] propose an extension of the capture and replay technique to increase efficiency, by capturing a minimal subset of the state of the application and environment suitable for replay. The increased efficiency results from decreasing the overhead of the capture and replay method at execution time.

2.1.5. StEP Method

The Stubbed Energy Profiling (StEP) method enables software producing organizations to "create an in-depth energy profile of their software products" [1]. The method measures energy consumption of software elements by comparing a benchmarked version of the software product to a version where some selected software element is stubbed. The difference in energy consumption between the benchmarked version and the stubbed version indicates the software energy consumption of the software element. The StEP method is applied manually by Jagroep *et al.* in their evaluation of the method.

Besides measuring the difference in energy consumption between the benchmarked and stubbed versions of the software product, the StEP method consists of eight activities in total [1]. The final three activities are repeated for each of the functional elements selected during activity 2. Figure 2.1 illustrates the StEP method activities.

1. Select the functional elements that should be prioritized for testing, in relation to an energy requirement of the software architecture;
2. Adapt the architectural description to identify the places where the stubbed software elements are required, and to identify what stubs can be created for the different functions of software elements;
3. Create test case(s) that execute the functional elements identified in the previous step. Where possible, (re)use existing tests to minimize the impact of applying the StEP method on the software development process;
4. Determine measurement approach based on the test cases(s), and identify the metrics that should be measured to quantify the energy requirement;
5. Profile the benchmark version of the software product by performing the test case(s) on the non-stubbed version;
6. Introduce the stub into the software product by stubbing an identified functional element;
7. Profile the stubbed version in the same manner as the benchmark version was pro-

filed;

8. Annotate the architectural description by including the energy consumption that was caused by stubbing a functional element.

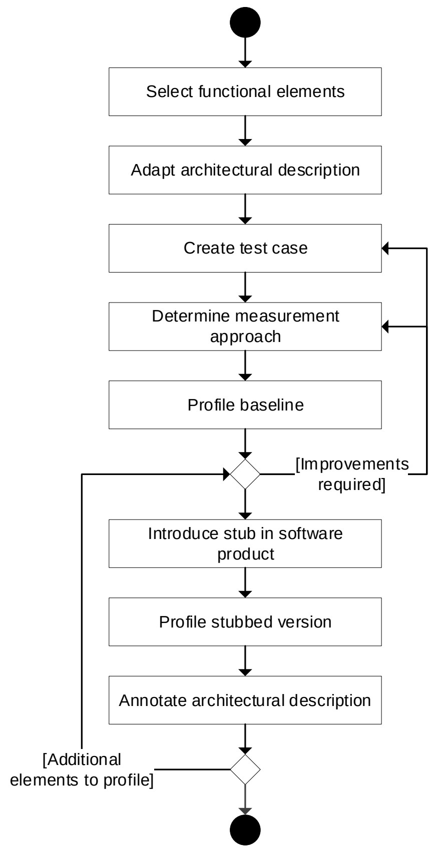

Select functional elements
Adapt architectural description
Create test case
Determine measurement approach
Profile baseline
[Improvements required]
Introduce stub in software product
Profile stubbed version
Annotate architectural description
[Additional elements to profile]

Figure 2.1.: StEP method, after [1].

2.2. The C# Programming Language

In this section we introduce the C# programming language, the .NET Framework on which programs written in C# run, and the compiler platform of the .NET Framework. We use the C# 6.0 specifications when referring to concepts of the C# programming language. However, as the C# 6.0 specification [29] is a draft at the time of writing, we refer to the C# 5.0 standard specification [30] whenever concepts have not changed between the versions of the specification.

2.2.1. Language Concepts

Executable code in C# programs is contained in assemblies: "one or more files output by the compiler as a result of program compilation" [30]. Assemblies are either *class libraries* (`.dll` which can be used by other applications), or an *application* (`.exe`) [30].

A C# program is organized using namespaces, which can contain nested namespaces and *type declarations* [30]. A type declaration defines a class, struct, interface, enum, or delegate [30]. Types are either *reference* types or *value* types [30, Ch. 9]. Value types directly contain their data, whereas reference types store references to their data [30]. We refer to *objects* when discussing structs and classes. Struct type declarations are value types, and class type declarations are reference types [30].

Depending on the type declaration, a type declaration can contain various kinds of members. Members of classes and structures are either *data* members, which store the data contained in the type, or *function* members, which contain executable statements [30]. Moreover, members can be declared as a *static* member or an *instance* member [30]. Static members are declared with a `static` modifier, and are bound to the type declaration; they are shared among all instances of the type declaration. Instance members are bound to instances of object types, and operate on a specific instance thereof.

Whether or not a member declaration is accessible from other parts of the program is determined by the *declared accessibility* of the member [30]. The declared accessibility is determined by one of the following access modifiers on the member declaration: `public`, `private`, `protected`, `internal`, or `protected internal`.

Type declarations can be split over multiple C# source files by including the `partial` modifier to each part of the partial type [30]. The partial definitions of a type declaration are merged at compile-time.

2.2.2. Memory Management

Memory is managed automatically for C# applications, which is implemented by a garbage collector [30, Sec. 8.9]. The garbage collector manages the allocations of objects, and reclaims the memory of objects that are no longer used by the application [31]. Allocations and deallocations of objects are performed in a managed heap in memory. Garbage collection is performed in two parts [32]:

1. *Garbage detection*: determine which objects in memory are *live*—allocated objects reachable by some path of execution in the application code—and which objects are garbage;

2. *Garbage reclamation*: free the memory of objects that are no longer accessible by the application.

The garbage collector implements *generational collection* [31, 32]. The managed heap is divided in generations which separates objects by age. The garbage collector used by C# has three generations: generation 0, 1 and 2 [31]. Generation 0 contains the youngest and short-lived objects, and generation 2 contains the olders and long-lived objects. Generation 1 acts as a buffer between the short-lived and the long-lived objects. When an object is not reclaimed by the garbage collector, it is advanced to the next generation. Objects in the final generation remain in the final generation until they are reclaimed. Garbage collection is performed more frequently for young objects, as most objects live a very short time [32].

2.2.3. The .NET Framework

The .NET Framework supports the building and running of applications developed in C#, F#, Visual Basic, and C++/CLI [33]. The .NET Framework has two main components: the Common Language Runtime (CLR) [34], and the .NET Framework class library [33].

The Common Language Runtime manages and executes code written for the .NET Framework [35]. It is an implementation of the Common Language Infrastructure (CLI) [35], as specified in the ECMA-335 standard [36]. It provides services such as code execution, memory management, thread management, and exception handling [34].

The .NET Framework class library is a collection of object-oriented reusable types for the use with the Common Language Runtime [33]. It provides types commonly used when programming, such as: string management, data collection, serialization, and file I/O.

2.2.4. Roslyn: the .NET Compiler Platform

The Roslyn .NET compiler platform exposes the internals of the .NET C# and Visual Basic code analysis APIs [37]. It is extensively used for various Microsoft Visual Studio features such as: IntelliSense, code refactoring, and renaming [37]. Furthermore, it exposes an interface for interacting with MSBuild, the build tool used by Microsoft Visual Studio for compiling applications for the .NET Framework [38].

Roslyn is divided into four APIs, each which expose a subset of the code analysis functionality of the compiler [37].

- **Compiler APIs.** Exposes the syntactic and semantic objects models that the compiler uses internally.

- **Workspace APIs.** Organizes information about Visual Studio solutions, projects, and documents contained in such project. For example, a C# source file (`.cs`) is a document that is part of a project.
- **Diagnostic APIs.** Information emitted by the compiler about warnings, errors and improvements found during compilation of the source code of programs. Allows user-defined analyzers to plug-in into the compiler, and produce user-defined diagnostic information.
- **Scripting APIs.** Allows hosted runtime execution of snippets of C# and Visual Basic in user programs.

Roslyn performs compilation in a three-step process [39]:

1. Parsing of source code files into syntax trees (*syntactic model*);
2. Binding source code elements to symbols (*semantic model*);
3. Emitting the compiled code in the Intermediate Language (IL) format to an assembly.

Syntax Model

The syntactic model of a program is represented by a syntax tree. A syntax tree consists of three distinct elements: syntax nodes, syntax tokens, and syntax trivia [37]. Syntax trees hold all information about the source code, including errors found during parsing, and can be transformed back exactly to the original source code [37].

Syntax nodes are the non-terminal nodes of syntax trees. They represent syntactic information, such as declarations, statements and expressions [37].

Syntax tokens are the terminal nodes of syntax trees. They represent syntactic terminals, such as keywords, identifiers and literals [37]. Terminals are the elemental symbols of a language as defined by its grammar [40].

Syntax trivia are parts of the source code which are not part of the language syntax [37], such as comments and whitespace.

Semantic Model

The semantic model of a program is represented by its compilation. A compilation contains all (external) references, compilation options, and source code files [37]. Additionally, a compilation stores a set of symbols.

A symbol represents semantic information about some element declared in a source file or referenced from an imported assembly [37]. Furthermore, symbols contain additional information determined by the compiler during compilation; where it is defined in the source code; and, can be used to find references between symbols.

Chapter 3

Research Approach

The research that we present is divided into three phases. First, a software tool is developed to automatically generate stubs of functional components for software products for which we want to determine the software energy consumption. Second, a set of experiments is performed to show the feasibility of measuring software energy consumption using the StEP method and stubs generated by our software tool. Third, a case study is performed with a software product from the case company to show that, by improving the StEP method using automatically generated stubs, we can reduce the effort required to apply the StEP method. The experiments and case study follow the guidelines provided in [41].

In this chapter we discuss the research approach that we use for the research presented in this Master's Thesis. Section 3.1 formulates the research questions that we aim to answer. Section 3.2 discusses the hardware setup that is used to perform the experiments and case study to measure the software energy consumption. In Section 3.3 we discuss the method that we apply to measure software energy consumption.

3.1. Research Questions

This research focuses on automating the measurement of the energy consumption of software architectural components as defined in the StEP method, as discussed in Section 2.1.5. As noted by Jagroep *et al.* [1], the creation of stubs in the StEP method remains time consuming and is subject to improvement. We aim to automate the creation of stubs by applying the Capture and Replay method as discussed in Section 2.1.4. We formulate a research question (RQ).

RQ1: How can we automatically measure the energy consumption of software architectural components within the software development process by stubbing?
Measuring the energy consumption of software architectural components remains a time consuming and manual effort. By applying the Capture and Replay method to the StEP method, and studying its effects in relation to the software energy consumption, we aim to reduce the manual effort.

The Capture and Replay method is chosen based on the software developed at the case company, and based on the software development process used at the case company. They are familiar with tools which apply the Capture and Replay method to test the software applications under development. In consulation with the case company, we determined the Capture and Replay method to be the most feasible approach for automating the creation of stubs.

To answer the research question, six sub questions (SQs) are formulated. Each question is individually addressed below.

> **SQ1: How can we automate the creation of stubs in the StEP method by applying the Capture and Replay method?**
> Applying the capture and replay method requires the modification of software, which can be done using a variety of techniques. Furthermore, both the capture and replay phase are configurable in terms of what to capture, how to store the captures, and how to play them back.

> **SQ2: How do we isolate the software architectural components to measure its energy consumption as accurately as possible?**
> The energy consumed by a software architectural component is not necessarily confined to the component itself. The component might cause side effects, where energy consumption changes happen outside of the reach of the software architectural component. Measuring energy consumption accurately requires an isolation of where the energy consumption happens.

> **SQ3: How do we measure the impact of instrumentation on the energy consumption?**
> Instrumentation can be added to existing source code or programs by various means of modification. Adding instrumentation introduces an overhead on energy consumption in the energy profile of the program. To accurately measure the energy consumption of a software architectural component the overhead introduced by instrumentation must be identified.

SQ4: How do we measure the impact of the replay on the energy consumption?
The replay of the captured interactions consumes energy by reading from the log. Measuring the energy consumption of the capture phase is possible, by comparing the unmodified program to the program with the capture phase built in. For the replay phase there is no benchmark to compare against, hence a different approach is required to gain insight into the energy consumption of the replay phase.

SQ5: How do side effects of components influence the energy consumption during capture and replay?
Side effects of the software architectural component that is being investigated can indirectly lead to changes in energy consumption of other software architectural components, or of different systems. For example, when the software architectural component uses a database that is deployed on a different server.

SQ6: What are effective ways to show the measured energy consumption during the software development process?
The StEP method communicates the measured energy consumption of the selected functional elements by annotating the architectural description. Making such an annotation is often done manually. Providing direct feedback of the energy consumption changes might be better suited to fit the automated process.

3.2. Experiment Setup

The experiment setup is built according to the guidelines of Jagroep *et al.* [1, 16]. Table 3.1 summarizes the specifications of the systems available for the research. A total of six servers are available, three HP DL380 G5 servers and three HP DL360 G7 servers. Depending on the requirements of experiments to be performed, the HP DL380 G5 servers, the HP DL360 G7 server, or a combination of both servers can be deployed for the test and control servers. Two Watts UP? PRO (WUP) meters are available to perform power measurements, that can be placed between the power lines of the test servers. Furthermore, a laptop is available for development and remote control of the server systems.

	Laptop	Server	Server
Model	Dell Latitude 5580	HP DL380 G5	HP DL360 G7
Operating System	Windows 10	Windows Server 2008	Windows Server 2016
Processor	Intel Core i5-7200U	Intel Xeon E5335	Intel Xeon E5645
Memory	8 GB DDR4-2400	8 GB FB-DDR2	42 GB, 36 GB, 24 GB DDR3-1333

Table 3.1.: Specification of systems available for research.

3.3. Energy Consumption Measurement approach

Profiling a software product to measure software energy consumption requires a measurement approach to ensure consistency. Following the methods described in [1, 16], the measurement approach consists of five aspects.

Energy consumption measurements

Energy consumption measurements are taken using a combination of hardware and software measurements. A Watts UP? PRO (WUP) power meter device is used to perform the hardware measurements. Microsoft Joulemeter energy profiler performs the software measurements on the test servers.

Performance measurements

Hardware performance is recorded to accurately relate energy consumption measurements to individual software elements [42]. Following the *Unit Energy Consumption* definition [16], the following hardware resources will be monitored using performance counters:

- CPU: utilization %.
- Memory: working memory bytes.
- Disk: bytes read/sec, bytes write/sec.
- Network: bytes sent/sec, bytes received/sec.

Idle energy consumption and cooldown time

Calculating the software energy consumption requires us to know the idle energy consumption for the hardware [42]. The idle energy consumption needs to be determined for each piece of hardware, by performing energy consumption measurements on the hardware that is running without any active software. Measurements for the idle energy consumption need to include the measurement software, as the measurement software itself causes energy consumption as well.

Furthermore, the cooldown time of the systems after a reboot need to be determined [42]. After a reboot, various processes and services related to the operating system are active on the systems. The energy consumption of these processes and services would pollute the energy consumption measurements. The period until when these processes and services become inactive is the cooldown time.

Data synchronization

Energy consumption and performance measurements are taken from multiple systems. For the measurements to be compatible, they need to be synchronized. Synchronization of the clocks of the systems is done using the Network Time Protocol (NTP).

Measurement protocol

To perform reliable measurements and ensure consistency between measurements we use the following measurement protocol, based on the measurement protocol followed by Jagroep *et al.* [1]:

1. Restart the test server.
2. Close unnecessary applications, services and processes.
3. Remain idle for the duration of the cooldown time.
4. Start Watts UP? PRO, Joulemeter, and performance measurements.
5. Start test and wait for test to finish.
6. Collect data.

Chapter 4

Automatic Creation of Stubs

The effort required to apply the StEP method is one of its largest limitations [1]. A major contributor to this limitation is the creation of the required stubs. Depending on the functional element that is to be stubbed, creating an implementation for the stubs could become very complex.

The effort required to apply the StEP method can be reduced by automating the creation of stubs, thereby tackling one of the largest limitations of applying the StEP method. The goal of this chapter is to answer the first research sub question.

SQ1: How can we automate the creation of stubs in the StEP method by applying the Capture and Replay method?

This chapter is organized as follows. Section 4.1 introduces the approach that we take to automate the creation of stubs. Section 4.2 explains how we apply the Capture and Replay method within the presented approach. Sections 4.3 to 4.6 discuss the implementation of the Automated Stubbing tool.

4.1. Approach

To automate the creation of stubs we apply the *Capture and Replay* method [24, 28]. Given a selected functional element for which we want to identify the software energy consumption, as determined in the StEP method, the Capture and Replay method is used to generate stubs by building two versions of the selected functional element. First, we create a version where the selected functional element is modified by inserting instrumentation statements for the capture of program traces. Second, we create a version where the selected functional element is replaced by a functional element which exposes an identical interface, but which replays the captured program traces.

Four versions of the program play a role in determining the software energy consumption:

Baseline version. The unmodified version of the program, which provides the baseline for the software energy consumption measurements.

Capture version. The program modified for capture of the interactions of the selected element.

Replay version. The program modified for the replay of the captured interactions.

Baseline+Replay version. The program modified for the replay of the captured interactions, whilst preserving the computations of the baseline version.

By measuring and comparing the software energy consumption of the four versions of the program we can determine the software energy consumption of the selected functional element. Compared to the StEP method, an extra step of profiling needs to be performed. We discuss the method of acquiring and computing the software energy consumption when using the Automated Stubbing method in Chapter 5.

4.1.1. Assumptions

The Automated Stubbing approach that we present works under the following assumptions.

First, we assume that side effects of interactions are limited to the state of the selected functional element. We can inspect the interal state of the selected functional element, however, it is not straightforward to determine which side effects happen outside of the state of the selected functional element. We leave it up to the users of the Automated Stubbing approach to identify suitable functional elements with limited side effects.

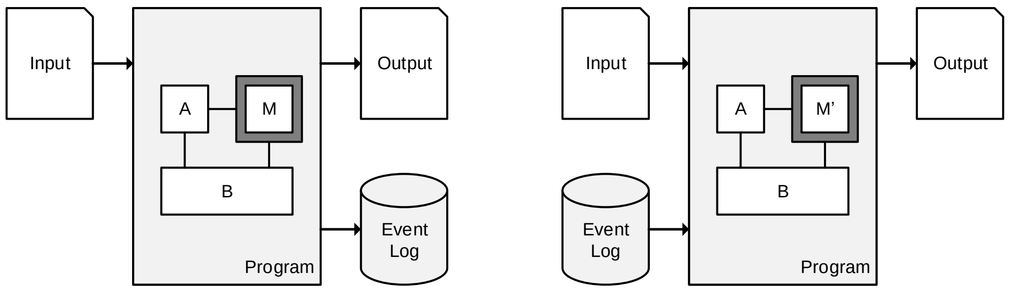
Second, we assume that the order in which the interactions take place that are captured and replayed is deterministic. Concurrent interactions which result in non-linear orders are not supported. The replay performs the captured interactions in the same linear order as they were captured. Would concurrent interactions cause a different order, then the replay run will be ignored. This simplifies the replay of the interactions, and allows us to verify that we are replaying the same execution of the captured interactions over multiple replay runs.

Third, we assume that the replay stub of the selected functional element removes any heavy computations from the element. The resource utilization and software energy consumption of the replay stub is expected to be significantly lower compared to the resource utilization and software energy consumption of the Baseline and Capture versions of the program.

4.2. Using Capture and Replay to Automatically Create Stubs

The Capture and Replay method consists of the capture phase and the replay phase [28]. First, the capture phase monitors the interactions between some selected functional element and its environment during the execution of the software system. Based on the monitored interactions a stub of the selected functional element is generated. Second, the stub of the selected functional element replays the captured interactions.

Figure 4.1 illustrates the Capture and Replay method. Within the program, element *M* is some selected functional element for which we want to capture and replay its interactions. *Interactions* are sequences of program instructions that expose a set of functionalities within the program. Element *M'* is the stub of the selected functional element *M* modified for replay of the captured interactions. The dark grey area around *M* and *M'* is the boundary on which the interactions between the element and its environment take place. The *environment* consists of all functional elements of the program, including all functional elements in standard and third-party libraries, besides the selected functional element *M*. The *input* of the program is provided by a test case. The *event log* stores all captured interactions, which can later be retrieved for replay.



(a) Capture phase **(b)** Replay phase

Figure 4.1.: Capture and replay method, after [28].

Both the capture phase and the replay phase consist of three steps [28]. The first two steps in both phases are equivalent. The capture and replay method is performed for a selected functional element under test.

1. Identify the interactions between the element and its environment;
2. Instrumenting the application code;
3. Capture or replay interactions between the element and its environment:
   a) Capture interactions at runtime, and output captured events to an event log.
   b) Replay captured interactions from an event log.

The Automated Stubbing tool implements the steps of the capture and replay phases for the automatic creation of stubs. The final step of the capture and replay method is performed when the profiling of the stubbed version of the application is performed, as part of the StEP method.

4.2.1. The Role of the Event Log

Multiple approaches can be taken to implement the event log. Most importantly, the interactions that are captured have to persist between the capture phase and the replay phase. The approach taken during capture may differ from the approach taken during replay.

For example, the captured interactions can be stored in a file on disk, or in some database. During replay the interactions can be replayed by providing a replay scaffolding — interactions are collected from the event log at runtime and replayed accordingly [28] — or by implementing a mock object [24] in which the event log is compiled as a lookup table as part of the object.

The Automated Stubbing method presented in this chapter uses the replay scaffolding approach. Replay scaffolding allows for more flexibility compared to a mock object. First, capture and replay can be performed for many different inputs without having to recompile the stubbed versions of the application. The replay scaffolding replays the events from the event log regardless of how many captures are performed, whereas a new mock object is to be created if the capture differs. Second, even though a mock object implementation can provide a lookup in $O(1)$ time — e.g. when using a perfect hash function — creating such a mock object can become difficult when dealing with increasingly large event logs. If the size of the event log becomes too large, the compiler might not be able to produce an assembly.

4.3. The Automated Stubbing Tool

The method proposed in this chapter is implemented in an automatic stubbing tool developed in C#, and supports the automated creation of stubs for programs written in the C# programming language. The tool implements the process shown in Figure 4.2. Based on the selected functional element, the interactions of the functional element and its environment are identified. The identified interactions are instrumented for capture and for replay, resulting in two new versions of the program. Finally, the instrumented versions of the program are compiled and output to their corresponding assemblies.

The automatic stubbing tool consists of three components: a command line application

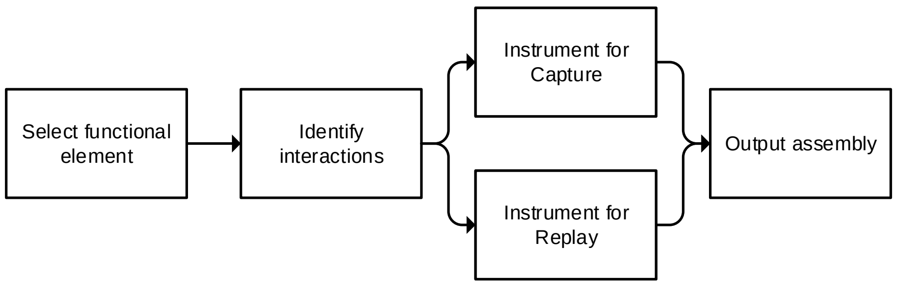


Figure 4.2.: Automated Stubbing tool process.

which can be used to create the stubbed versions of a given program (Section 4.4); a class library which implements the interface into the event log (Section 4.5); and, a class library which implements the instrumentation logic (Section 4.6).

4.4. The Automated Stubbing Tool: Command Line Application

The command line application (CLI) is the primary interface of the Automated Stubbing tool. Given a Visual Studio C# project file and the fully-qualified name of some object type within the project file, it generates the Capture and Replay versions of the assemblies defined by the project file. The CLI instruments the object type where required, and invokes the MSBuild program to build the executable assemblies.

The CLI depends on the class libraries which implement the event log interface, and the instrumentation logic. Furthermore, the connection with the event log interface from the Capture and Replay versions can be configured by the CLI.

4.5. The Automated Stubbing Tool: Event Log Interface

The event log class library exposes the interface that the Capture and Replay versions of the applications use to communicate and serialize with the event log. The class library contains data types which specify how interactions are encoded. Furthermore, the class library contains an interface which allows the capture and replay of interactions.

In this section we discuss which interactions we identify, and how we encode these interactions.

4.5.1. Interactions

Interactions are sequences of program instructions that expose some set of functionalities within the program, which take place between the selected functional element and its environment [28]. For example: function calls, field accesses, and thrown exceptions.

We distinguish between ingoing interactions, and outgoing interactions [28]. *Ingoing* interactions are interactions that originate from the environment, and request some functionality from the selected functional element. *Outgoing* interactions are interactions that originate from the selected functional element, and request some functionality from the environment.

Interactions in C#

Capturing interactions between the environment and the selected functional element, and vice versa, limits the capture of interactions of C# object members which are accessible by other types. We refer to the accessible members which are candidates for interactions as *exposed members*. Exposed members have a declared accessibility of: `public`, `protected`, `internal`, or `protected internal`. Members with a declared accessibility of `private` are not captured, as these occur only within objects themselves and are implicitly captured and replayed.

Furthermore, capture and replay is performed for function members only — object members that contain executable statements. Field accesses are not captured, as our approach captures the internal state of the object within each method interaction, and thereby keeps track of changes to fields of the objects. C# has the following function members [30]: methods, properties, events, indexers, user-defined operators, instance constructors, static constructors, and finalizers.

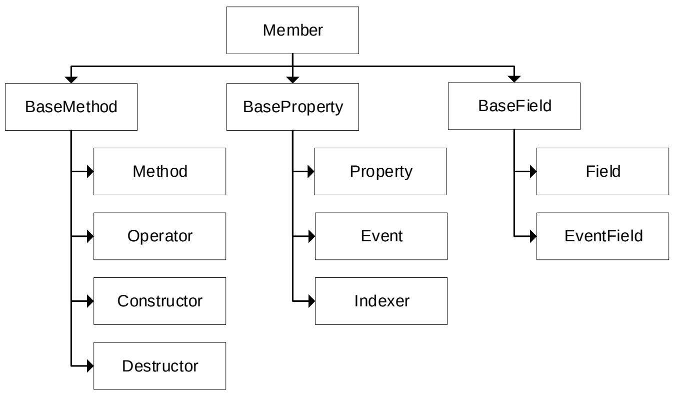


Figure 4.3.: C# field and function members syntax relationships.

Figure 4.3 illustrates the relationships between the C# function and field members as part of the syntactic model, and the base syntax declarations which contain the common abstractions.

Properties, events, and indexers share the same base type: *BaseProperty*. They share two common properties [30]: a field tied to the property, and a set of accessors containing executable statements. Properties and indexers use accessors to *get* and *set* the value of the field associated with the property or indexer. Events use accessors to *add* or *remove* delegates to notify when the event is fired. Delegates are types which references methods with a given return type and parameter list [30] The accessors can be implemented by developers, or are generated automatically by the compiler. The compiler adds a new field to the object of which the property is a member when it generates accessor methods. Therefore, a property in C# can be seen as a set of methods which is tied to some field.

Methods, operators, constructors, and destructors share the *BaseMethod* base type. They share three common properties [30]: a body of executable statements, a set of parameters (which can be empty), and optionally a return expression.

The Atuomated Stubbing tool only captures method interactions. Because methods, operators, constructors and destructors share the same base type; and properties, events and indexers are a combination of a set of methods which are tied to some field.

4.5.2. Encoding Interactions

For each interaction that we capture and replay, we encode information to be able to uniquely identify interactions.

Within interactions we distinguish instances of objects by using an *instance ID* to uniquely identify object instances. The instance ID functions like the object ID described by Orso *et al.* [28]. Instance IDs are implemented using a global numeric counter. When capture or replay starts, the global counter is initialized to zero. Every time a object instance is created, the instance is assigned an instance ID from the global counter. Furthermore, static instances and null references use predefined unique instance IDs, which are assigned negative numbers internally.

Objects require an additional field to track the instance ID of the instances of the object. However, within applications instrumented by the approach many functional elements and their objects are not instrumented by our method, or cannot be instrumented. For example, objects found in the .NET Framework library cannot be instrumented by our method. The instance IDs of these objects are tracked using an *instance map*: a key-value map which tracks which instance ID belongs to a given object reference [28]. To not extend the lifetime of the objects stored in the instance map, which in cases could lead to the program running out of memory, the object references are stored as *weak references*. A weak reference holds a reference to some object, but allows the object to be collected by the garbage collector [39, Ch. 12].

Method Interaction

Method interactions are implemented in C# as *calling sequences* and *return sequences*, where code is divided between the calling method (*caller*), and the method that it calls (*callee*) [40, Sec. 7.2.3]. We identify two kinds sequences that we encode: *method entry* and *method exit*. Method entry is the calling sequence that takes place when code execution moves from the caller to the callee. Method exit is the return sequence that takes place when code execution moves from the callee back to the caller.

For a method interaction we encode: the *instance ID* of the instance to which the method belongs, the *signature* of the method, the *parameters* passed to the method, a key-value map which stores the *state* of the containing object at the moment of capture, and optionally the *return value*.

The signature includes the fully-qualified name of the method, including its containing namespaces and classes, and the types of the parameters [30, Sec. 8.6]. We do not capture the return type of the method, as C# does not allow polymorphism on the return type [30, Sec. 8.6]. Therefore, the return type is given by the signature of the method.

Parameters

For a parameter we encode: the *type*, the *name*, and the *value* passed to the parameter.

We distinguish between parameters that have a value type, and those that have a reference type. The values of value types are serialized and stored as the value of the parameter. For reference types we resolve an instance ID of the referenced instance, which is stored as the value of the parameter. Reference types are not serialized to limit the overhead of serialization, which time-wise can become as large as 500 % [28].

Object State

The state of the object is encoded as a map of key-value pairs, which we refer to as the object *state map*. The state map is used to recreate side effects that occur as a result of interactions that we capture and replay for the selected functional element. Each field of the object is stored in the state map, except for fields having a `const` or `readonly` modifier. Fields with a `const` or `readonly` modifier cannot be changed during the execution of the program, and thereby do not change when capturing and replaying the selected functional element. A field is stored in the map as a key-value pair with the name of the field, and its value.

Storing the state of the object in a state map limits the types of objects which we are able to capture and replay. As we store the values of fields, the types of the fields have to be *serializable*. Furthermore, we require the serialization to be automatic, i.e. no modifications to types are required to perform serialization.

The .NET Framework class library includes three serialization mechanisms: data contract serialization, binary serialization, and XML serialization [39]. Of the three serialization mechanisms the *binary serialization* is highly automated, and implemented for many of the types of the .NET Framework class library [39]. Binary serialization is supported for types that have the `Serializable` attribute, or implement the `ISerializable` interface. Binary serialization provided by the .NET framework for types that have the `Serializable` attribute is limited to the serialization of public and private fields, and public properties [39].

However, not all user defined types that we encounter during capture and replay are automatically serializable. We do assume that all types that are stored in the state map are serializable, and we leave it up to the developer the implement the serialization if required.

4.5.3. Event Log Interface

Executing the program modified for capture, the capture instrumentation generates events from the captured interactions and writes these to the event log. Executing the program modified for replay, the replay instrumentation collects events from the event log. Based on the collected event the captured interaction is replayed. The execution depends on the input provided for the program, which should execute the selected functional element for which we determine the software energy consumption.

The event log can be stored on a remote server, to minimize the impact of the event log on the software energy consumption of the server which runs the program for the experiments. The event log is implemented using MongoDB. MongoDB is a NoSQL database, which stores data in documents — dynamic semi-structured data structures represented in a JSON-like format [43]. The document model used by MongoDB allows us to map C# objects directly to documents stored in MongoDB. The flexibility provided by the document model enables us to store and retrieve the encoded interactions as required.

The MongoDB document model stores the documents in the BSON format[1]. The BSON serialization implemented by the MongoDB class libraries works on the same principles

[1]`http://bsonspec.org/`

as the serialization mechanisms of the .NET Framework class library. However, the BSON serialization requires C# object types and their members to be mapped to BSON documents, which is referred to as *class mapping*[2], The class mapping process is automated, but most user defined types must explicitly be registered to implement serialization. We leave it up to the developer to register the class maps of types if required.

4.6. The Automated Stubbing Tool: Instrumentation Logic

The instrumentation logic class library implements the generation of the Capture and Replay stubs, and outputs compiled assemblies of the stubbed versions of the application.

Appendix A contains source code examples of how the instrumentation is performed for various versions of the MD5 message-digest application of the experiments, which we discuss in Chapter 6.

This section is organized as follows. First, in Section 4.6.1 we discuss what instrumentation method we apply. Second, Section 4.6.2 discusses how the .NET Compiler Platform Roslyn is used to perform the instrumentation.

4.6.1. Instrumentation

To capture and replay interactions of the selected functional element, instrumentation needs to be added to the program and the selected functional element [24, 28]. Instrumentation is a method which inserts extra code in a program to observe its behavior [44]. Instrumentation can be performed by *source instrumentation*, or by *binary instrumentation* [44]. Source instrumentation modifies the behavior of the program before or during compilation, by transforming the source code of the program. Binary instrumentation modifies the behavior of the program of the compiled executable, which can also be done at runtime, by modifying the byte-code instructions.

The automated stub generation method presented in this chapter uses *source instrumentation* during compilation to add instrumentation to support capture and replay of a selected functional element. There are several reason why we chose to use source instrumentation instead of binary instrumentation. First, as discussed by Jagroep *et al.* the StEP method is applied during the development process of software [1]. As we are interested in determining the software energy consumption of a functional element from a software application during development, we have access to the source code of the software application. Second, binary instrumentation can be difficult to implement [45]. Compared to binary instrumentation, the source instrumentation can be modified and

[2]`http://mongodb.github.io/mongo-csharp-driver/2.5/reference/bson/mapping/`

extended by developers at the case company without the need of extensive knowledge of binary instrumentation and/or binary instrumentation frameworks.

4.6.2. Instrumentation with Roslyn

The Automated Stubbing tool uses the .NET Compiler Platform Roslyn (Section 2.2.4) to load C# project files, perform instrumentation, and emit executable assemblies. The instrumentation logic class library depends on two inputs:

1. A C# project file which defines the assembly that contains the selected functional element to stub;
2. A set of fully-qualified type names that determine the objects of the selected functional element.

In this section we discuss how we use Roslyn for the Automated Stubbing tool, by discussing the various parts of the instrumentation logic class library. First, we discuss the workspace abstraction used for managing the C# project file. Then we discuss the classes that implement the instrumentation logic for the automated building of stubs: the Stub Builder classes.

The Workspace

The workspace manages the C# project file, and enables the building of the stubbed versions of the assembly defined by the loaded project file. The workspace makes use of compiler API and workspace API of Roslyn.

Given the path to a C# project file, the project is loaded into an instance of the `Project` class of the workspace API. The `Project` class manages the project file, enables access to the documents that are part of the project, and gives access to the syntax trees and semantic models of the assembly.

After a C# project is loaded, the workspace uses the Stub Builders to generate the stubbed versions of the selected functional element. The workspace uses the semantic model from the `Project` instance to identify the objects and the function members of the objects, based on the given set of fully-qualified types names. The semantic model of a `Project` instance is contained in the compilation of the project. The compilation contains symbols which represent the semantic information of the source assembly. The semantic information is represented in a tree structure of namespaces, types, and type members.

C# objects can be defined in multiple source locations — multiple C# source code files — when the object is declared with the `partial` modifier. The object has a single symbol defining its semantic information, however, each source location has its own type declaration syntax node and subset of function members. The workspace invokes the Stub Builders mutiple times when the given object is defined at multiple locations. For each invocation, specialized Stub Builders use the type declaration syntax node and the subset of the function members — as syntax nodes that are defined at the partial source location — to perform the instrumentation.

The Stub Builder

The Stub Builder provides the interface used for the generation of stubs, and forms the base class for specialized Stub Builder instances which implement the generation of a specific kind of stub: Capture, Replay, or Baseline+Replay. The specialized Stub Builder instances will be discussed in the coming sections.

The core functionality provided by the Stub Builder is given by three sets of functions. Instrumentation is performed by modifying the syntax nodes of a C# object, and the syntax nodes of the function members of the object.

First, the Stub Builder provides an abstract method [39] to perform instrumentation for a method declaration. The specialized Stub Builder instances implement the abstract method to customize the instrumentation logic as required for the kind of stub.

Second, the Stub Builder has a single method which performs the instrumentation for a given object, and a set of function members of the object. We assume that the object instrumentation method can be called multiple times, as C# objects can be defined at multiple source locations. Each invocation is called with the object declaration syntax node of the object declaration of a single source location, and the set of function members to instrument. The set of function members are required to be children of the object declaration syntax node. We include the set of function members to give developers more options to configure the automated stub generation to their requirements. Depending on the requirements, not all function members might need to be instrumented. The object instrumentation method performs a two-step process to modify the object declaration syntax node:

1. Two local field declarations are added as private members to the object: a field for the connection with the event log, and a field which stores the instance ID of the object. If the object is declared at multiple source locations, the field declarations are added to one of the object declaration syntax nodes. Adding the field declarations multiple times to the same object would result in a compilation error.
2. For each given function member, the abstract method declaration instrumentation method of a specialized Stub Builder instance is invoked.

Third, the Stub Builder provides a set of methods which are common to the specialized Stub Builder instances. For example, the Replay Stub Builder and the Baseline+Replay Stub Builder both use methods to instrument method declarations for the replay of captured events.

The Capture Stub Builder

The Capture Stub Builder provides a specialized implementation to perform instrumentation for the capture version of a stub. It derives its basic functionality from the Stub Builder class, and implements the method declaration instrumentation logic for building stubs.

The Capture Stub Builder adds instrumentation statements to the body of the given method declaration in two locations: at method entry, and at method exit. At each instrumented location, part of the interaction with the method is captured, and an event is created and written to the event log. The instrumentation is performed as follows.

First, we add a statement to capture the parameters passed to the method at the beginning of the method body. The statement is only generated when the instrumented method has parameters. The captured parameters are used for both the method entry event, as well as the method exit event. Depending on the type of the parameter, we create an instance which references the parameter according to the encoding discussed in Section 4.5.2.

Second, to capture the method entry interaction we insert a set of statements after the capture of the parameters.

1. A state map of the object is created, capturing the current state at method entry;
2. A method entry event is created and added to the event log. The method entry event captures the method signature, the method parameters, and the object state.

Third, to capture the method exit interaction(s), each return statement is replaced to capture the return value of the method: a state map of the containing object is created, capturing the current state at method exit; the expression that is returned is captured in a temporary local variable; a method exit event is created and added to the event log; and, the temporary local variable containing the return expression is returned. If the method returns void, the return expression cannot be captured, and the temporary variable need not be created. Furthermore, methods that return void can have an implicit return statement at the end of the method body. The instrumentation is always added to the end of the method body for methods that return void, but do not have return statements as the last statement of the method body. The method exit event captures:

the method signature, the method parameters, the object state, and optionally the return expression.

The size of the instrumented code depends on the size of the state map — the number of fields of the object, and the amount of return statements present in a method body. Each return statement, including the implicit return statement that a method which returns void can have, requires the creation of the state map. Therefore, the size of the instrumented code can grow considerably large when dealing with objects with many fields, or when dealing with methods with many return statements.

The Replay Builder

The Replay Stub Builder provides a specialized implementation to perform instrumentation for the replay version of a stub. It derives its basic functionality from the Stub Builder class, and implements the method declaration instrumentation logic for building stubs.

The instrumentation for replay replaces the body of the method that is instrumented. The instrumentation is performed by inserting the following statements in the method body:

1. The parameters that are passed to the method are captured identical to how parameters are captured by the capture instrumentation. Clearly, no statement needs to be created if the method does not have parameters.
2. A method exit event is collected from the event log, given the object instance ID, the method signature, and the parameters captured in the first statement. If the event log does not find an event, an exception is thrown by the event log stopping the replay of events.
3. The state of the object is restored given the state map that is captured with the method exit event. This statement is not generated when the object does not contain any state members.
4. If the method returns some value, the return expression from the collected event is returned.

The Baseline+Replay Builder

The Baseline+Replay Stub Builder provides a specialized implementation to perform instrumentation for the Baseline+Replay version of a stub. It derives its basic functionality from the Stub Builder class, and implements the method declaration instrumentation logic for building stubs.

The instrumentation for the Baseline+Replay stub adds replay instrumentation to a method, while preserving the statements present in the method body. The instrumentation is performed as follows.

First, similar to the other Stub Builders, a statement that captures the parameters passed to the method is inserted as the first statement of the method body. The statement is not created if the method does not have parameters.

Second, at each method exit interaction a subset of the replay instrumentation is inserted. This is inserted at the same locations as where the Capture Stub Builder captures method exit interactions. A method exit event is collected from the event log, the state of the object is restored given the state map that is captured with the method exit event, and the return expression is returned (if the method returns some value). We return the return expression present in the unmodified method declaration, because the return expression could invoke methods performing computations that we want to preserve, or invoke methods which are instrumented by the Automated Stubbing method also.

4.6.3. Using the Semantic Model to Guide Instrumentation

The Stub Builder classes use the semantic model of the C# project to guide the instrumentation. Three parts of the semantic model are used: the fully-qualified names of types, the semantic information of types and type members, and the syntax trees attached to the semantic information.

First, the semantic model contains the fully-qualified names of all types that are part of the project, and that are referenced by the project. Given the set of fully-qualified type names that determine the objects of the selected functional element, the semantic model is used to gather the symbols that define the objects of the selected functional element.

Second, the information of the symbols that define the objects of the selected functional elements is used to identify the interactions which are candidates for instrumentation. Each symbol that defines a C# object contains the members defined by the object. Exposed members of the object are collected as a list of symbols. Together with the object symbol, the exposed members are passed to the Stub Builder instance to perform the instrumentation for an object.

Third, the specialized Stub Builder classes use the semantic information in the symbols of objects and object members. The symbols are used to extract the syntax trees for which the instrumentation will be performed. A single symbol may refer to multiple syntax trees, because C# objects may be defined in multiple source locations with the `partial` modifier. Furthermore, the fully-qualified names of types, and the signatures of members provided by the symbols are used to encode the interactions.

Chapter 5

Profiling

To measure the software energy consumption (SEC) [16] of a software application when applying the Automated Stubbing method, multiple versions of the software application need to be profiled. Compared to the StEP method — which compares the SEC of the unmodified application to the software energy consumption of the stubbed application [1] — by applying the Automated Stubbing method three versions of the stubbed application play a role in determining the SEC.

This chapter is organized as follows. Section 5.1 demonstrates how to apply the StEP method when using the Automated Stubbing method. Sections 5.2 to 5.4 discuss how to determine the SEC of the stubbed versions of an application when applying the Automated Stubbing method.

5.1. Extending the StEP Method

To determine the SEC of an application when using the StEP method combined with the Automated Stubbing method, four versions of the application need to be profiled: the Baseline version, the Capture version, the Baseline+Replay version, and the Replay version. Figure 5.1 shows the StEP method as extended with the Automated Stubbing method. We discuss the activities which are changed compared to the StEP method in detail.

Create Test Scenario

A collection of test cases which execute the selected functional elements should be designed in the *Create test scenario* activity. The test cases should induce varying levels of stress, to observe differences in the performance of the application [1]. The ability to

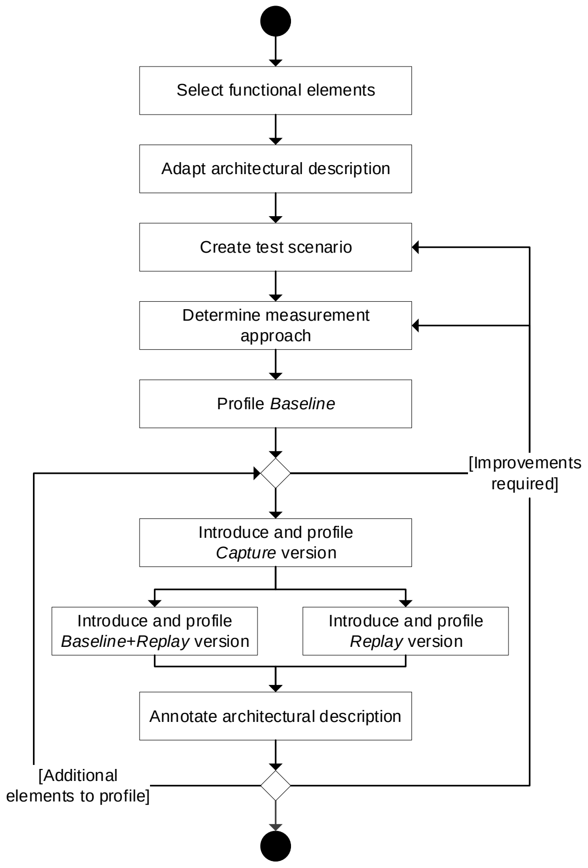

Select functional elements
Adapt architectural description
Create test scenario
Determine measurement approach
Profile *Baseline*
[Improvements required]
Introduce and profile *Capture* version
Introduce and profile *Baseline+Replay* version
Introduce and profile *Replay* version
Annotate architectural description
[Additional elements to profile]

Figure 5.1.: The activities of the StEP Method combined with Automated Stubbing.

automatically perform the test cases repeatedly is of importance as well, to exclude incidental findings caused by external factors and to gather a sample set of decent size. Furthermore, it is recommended to use realistic usage scenarios when designing the test cases [1].

Profile Baseline

First, the created test scenario is used to profile the unmodified application. The performance and SEC measurements should be used to validate the test scenario. Irregularities in the measurements of the Baseline version profiling can indicate that there are external factors which influence the measurements. For example, such irregularities may include a large standard deviation in the runtime of the test cases, or when the measurement results contain many outliers. Improvements to the test scenario, the measurement approach, or the experiment environment must be applied before proceeding with profiling of the stubbed versions of the application.

Introduce and Profile Stubbed Versions

Additional activities are performed compared to the StEP method to be able to get SEC measurements. Compared to the StEP method, when using the Automated Stubbing method multiple stubbed versions of the application are part of the computations for the SEC. We discuss the computations in Sections 5.2 to 5.4 in detail.

The *Introduce stub in software product* and *Profile stubbed version* activities of the StEP method are performed for each stubbed version of the application. We combined the two activities in a single *Introduce and profile stubbed version* activity. The combined acitivity is performed in order: first for the Capture version, then for the Baseline+Replay version, and finally for the Replay version. A capture must be performed before a replay can take place, as the replay requires an event log containing a sequence of captured interactions. However, the Replay version can be profiled before the Baseline+Replay version, considering that the Capture has already been performed.

The measurements of the profiled versions are used to compute the SEC of the functional element. To exclude irregular measurements, the profiling activities are performed multiple times to gather a reliable sample set of decent size. Individual review of incidental findings may be required to determine whether some external factor was at play, or whether the incidental finding was caused by the application under test. Validation of the profiling activities is therefore required, likewise to the validation performed during the profiling of the Baseline version.

5.2. Measuring Software Energy Consumption

The SEC — the measure for the total energy consumed by the software — is computed by subtracting the idle energy consumption from the total measured energy consumption [16] (Equation 5.1).

$$SEC = EC_{total} - EC_{idle} \tag{5.1}$$

The StEP method allows us to identify the SEC of a functional element of a software application [1]. By subtracting the SEC of a stubbed element from the measured baseline SEC, we get an indication of the SEC of the stubbed functional element (Equation 5.2).

$$SEC_{element} = SEC_{baseline} - SEC_{stub} \tag{5.2}$$

Valid comparisions between the SEC of different versions of an application, or of multiple profiling runs of a single application, can only be performed when the parameters supplied to the application, and the environment where the measurements are taken are consistent for each measurement sample. The hardware which runs the application must be the same for each sample, as energy consumption is dependent on the hardware in the environment [14, 46, 47]. The usage of different hardware configurations can have a significant impact on the SEC [47]. Furthermore, to ensure validity of the SEC measurements a measurement protocol should be followed [1].

5.3. Measuring Software Energy Consumption of Instrumentation

The Automated Stubbing method creates three stubbed versions of a selected functional element of a software application by applying the capture and replay method. The instrumentation required to perform the capture and replay lead to an increase of the SEC of the functional element. To give an accurate indication of the SEC of the stubbed version of the functional element when applying the Automated Stubbing method, the overhead of the capture and the overhead of the replay on the energy consumption need to be taken into account.

As the instrumentation performs additional work compared to the baseline of the application, we expect the stubbed versions of the applications to include an overhead of the energy consumption. The expected SEC of some application and its stubbed versions is shown in Figure 5.2.

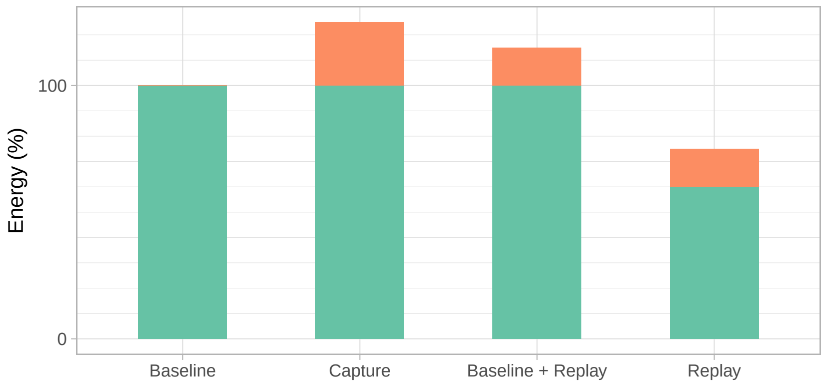


Figure 5.2.: Expected software energy consumption of the stubbed versions.

In this section we discuss how to determine the instrumentation overhead. First, we discuss how to determine the SEC of the capture instrumentation in Section 5.3.1. Second, we discuss how to determine the SEC of the replay instrumentation in Section 5.3.2.

5.3.1. Software Energy Consumption of Capture

The instrumentation added to the software application to perform the capture of interactions introduces performance overhead ,and changes the energy consumption of the application. Identifying the overhead of the capture instrumentation can be used to optimize the instrumentation statements, thereby reducing the overhead of the capture.

$$SEC_{capture\ overhead} = SEC_{capture\ stub} - SEC_{baseline} \tag{5.3}$$

The SEC of the capture instrumentation can be computed by subtracting the baseline energy consumption from the measured energy consumption of the capture version of the application (Equation 5.3). The instrumentation performs extra work compared to the baseline, therefore we assume that the energy consumption of the capture version of the application is always larger than the energy consumption of the baseline application.

5.3.2. Software Energy Consumption of Replay

Like the capture instrumentation, the replay instrumentation introduces performance overhead and the energy consumption of the application. Knowledge of the overhead of the replay instrumentation is required to acquire accurate measurements of the software energy consumption of the selected functional element. The energy consumption introduced by the replay is not part of the original functional module, and therefore should not be taken into account when calculating the SEC.

$$SEC_{replay\ overhead} = SEC_{baseline+replay\ stub} - SEC_{baseline} \tag{5.4}$$

The SEC of the replay instrumentation can be computed by subtracting the baseline energy consumption from the measured energy consumption of the Baseline+Replay version of the application (Equation 5.4). We use the Baseline+Replay version of the application, as this version performs the work of the Baseline version and the replay instrumentation. The Replay version of the application includes the overhead of the replay, but does not include the work originally performed by the application. Therefore, the Replay version of the application cannot be used to identify the overhead of the replay instrumentation.

5.4. Measuring Software Energy Consumption of the Functional Element

The SEC of the replay version of the application, which plays the role of the stubbed version of the application in the StEP method, provides an indication of the SEC of the functional element under study — which can be computed as shown in Equation 5.2. However, the replay version of the application includes the overhead of the replay instrumentation. The overhead needs to be taken into account to accurately determine the SEC of the functional element when applying the Automated Stubbing method.

$$\begin{aligned} SEC_{func} &= SEC_{baseline} - SEC_{replay} + SEC_{replay\ overhead} \\ &= SEC_{baseline} - SEC_{replay} + (SEC_{baseline+replay\ stub} - SEC_{baseline}) \\ &= SEC_{baseline+replay\ stub} - SEC_{replay} \end{aligned} \tag{5.5}$$

The SEC of the functional element when applying the Automated Stubbing method can be computed by taking the difference between the SEC of the Baseline and the SEC of the Replay stub, and adding the SEC of the replay instrumentation (Equation 5.5). The SEC of the replay instrumentation is added, because the Replay instrumentation overhead has a *negative* impact on the SEC. Without adding the Replay instrumentation, we underestimate the SEC of the stub.

The Baseline and Capture versions of the application should always be profiled, although the SEC of the element can be determined by profiling the Baseline+Replay version and the Replay version of the application. The profiling results of the Baseline and Capture versions of the application should be used to validate the created test scenario, validate the measurement protocol, and to ensure that external factors which could influence the profiling and measurements are under control.

Chapter 6

Experiments

To demonstrate the feasibility of using the Automated Stubbing method, a set of preliminary experiments is performed to determine the software energy consumption of a software application. The StEP method is applied in combination with the Automated Stubbing method to obtain the software energy consumption measurements, following the method outlined in Section 5.1.

This chapter is organized as follows. First, we discuss the application under test for the experiments (Section 6.1). Second, we discuss how the StEP method and the Automated Stubbing method are applied during the experiments (Section 6.2). Third, we show the results obtained from the execution of the experiments (Section 6.3). Fourth, we discuss the experiments, results, and findings (Sections 6.4 and 6.5).

6.1. Introduction

The experiments test the performance and software energy consumption (SEC) of an application which implements the MD5 message-digest algorithm to compute a hash value for a given input. The MD5 message-digest algorithm takes an input of arbitrary length, and computes a 128 bit message digest [48]. The MD5 message-digest algorithm is chosen because of readily available open source implementations, a set of computations which are suitable for stubbing, and an internal state which is updated throughout the rounds of the message-digest algorithm.

An open source implementation of the MD5 message-digest algorithm is used, provided by the Bouncy Castle Crypto package for C#[1]. The interface provided by the open source implementation is extended to provide an easy to use interface to compute hashes of

[1]The open source project page can be found at `https://github.com/bcgit/bc-csharp/`.

textual data using the MD5 message-digest algorithm.

6.2. Applying the StEP Method

To determine the SEC of the MD5 message-digest application the StEP method is applied, in combination with the Automated Stubbing method. In this section we discuss how we approach the activities of the combined method, following the process outlined in Section 5.1.

6.2.1. Select Functional Elements

The MD5 message-digest application computes the MD5 message-digest as a string representation for given textual data as input. The core functionality of the application is contained in a single functional element, which computes the MD5 message-digest from an array of bytes. The input textual data is encoded as an array of bytes before it is passed to the message-digest function. We formulate the requirement to determine the SEC of the MD5 message-digest application while performing its core functionality.

The MD5 message-digest functionality is implemented in a single C# class: `MD5`, which exposes a single functional member used to compute the MD5 message-digest for a given byte array of input. Thus, the selected functional element which will be stubbed is the `MD5` class.

6.2.2. Adapt Architectural Description

The MD5 message-digest application implements a single core functionality, therefore it does not have an architectural description. In the remainder of the chapter we use the selected functional element as well as the MD5 message-digest application to refer to the core functionality exposed by the application. The SEC of the selected functional element that we compute will be presented in tables.

6.2.3. Create Test Scenario

To test the core functionality of the MD5 message-digest application, a test scenario is designed which varies the load on the application. The application is stressed by varying the size of the input, where larger input sizes induce a larger load on the application.

To get reliable measurements the application needs to run for at least one second, because the Watts UP? PRO (WUP) power meter and Joulemeter energy profiler sample power measurements once a second. Would the application complete its computations within one second, the power measurements might not record the load created by the application as it falls between two measurement samples. The same holds for the performance measurements, because the Performance Monitor application takes performance samples of the system once a second. Initial testing of the application showed that to get reliable measurements, the minimum size of the input is 32 MB. The application computes the MD5 message-digest of an input of 32 MB in about one second. The minimum input size was determined by running the application with small amounts of input, and measuring the execution time. The smallest input size which would stress the application for at least one second is chosen.

To obtain a reliable data set from the experiments, each combination of the application version and input is executed 10 to 20 times. A cool down time of 15 minutes is used between each individual run. Each combination is executed a limited number of times due to the limited time available in the experiment environment to perform the experiments. For example, for a single test case which is repeated 10 times for each of the four versions of the application, the experiments take at least 10 hours ($10 \times 4 \times 15\,\mathrm{min} = 600\,\mathrm{min}$) to complete as result of the cool down time between the runs, not including the time it takes for the application to complete the test case.

6.2.4. Determine Measurement Approach

The energy consumption measurements are performed in the experiment environment, presented in Section 3.2. Following the measurement protocol outlined in Section 3.3, and the guidelines presented in [1, 42].

Energy Consumption and Performance Measurements

Energy consumption measurement data is collected using hardware measurements and software measurements. The hardware measurements use the WUP power meter device, the software measurements use the Microsoft Joulemeter energy profiler. Microsoft Performance Monitor is used to measure the performance of the test server. The clocks of the different systems are synchronized using the Network Time Protocol, to be able to relate events that occur on different systems.

Figure 6.1 illustrates the experiment environment. The servers used in the environment are HP DL380 G5 servers. The application server hosts the MD5 message-digest application, the management server hosts the MongoDB server instance, and the logging server is used to collected the power measurements from the WUP power meter device. A separate server is required to minimize the impact of the logging, and of the measurement collection on the energy consumption measurements. The dashed lines indicate

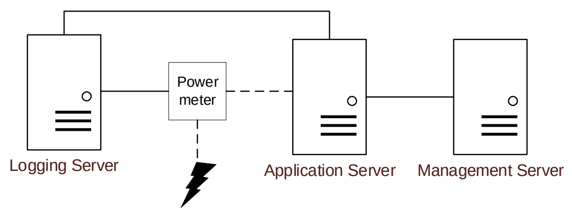


Figure 6.1.: Experiment environment.

power cables, the lightning bolt is the power source. The solid lines are network and USB cables used to transfer data between the systems.

Software Energy Consumption

The SEC of the application is computed by subtracting the idle energy consumption from the energy consumption measured during the experiments, following from Equation 5.1. The SEC is calculated by taking the average power between two measurements multiplied by the time between measurements, and taking the sum over the duration of the measurements.

Computing the SEC of the experiments requires the idle power consumption of the test servers. The idle power consumption is determined by measuring the power consumption of the hardware running without any active software [42]. The idle energy consumption of the servers is established at 249.49 W.

Measurement Protocol

The measurement protocol presented in Section 3.3 is used to perform the experiments. The experiment test runs are automated by usage of a batch script. The script repeatedly performs the following sequence of actions, following the profiling order of the StEP method combined with the Automated Stubbing method:

I. Record the start time of the execution;
II. Execute the given version of the application with the corresponding load;
III. Record the end time of the execution and write to the log file;
IV. Wait for the specified cool down time (15 minutes).

The time stamps in the log files are used to determine the duration of runs, and to relate SEC and performance measurements to specific runs.

6.2.5. Profile Baseline

The Baseline version of the application is profiled using the created test scenario, and the determined measurement approach. The collected SEC and performance measurements are used to verify the created test scenario. The profiling results, and the discussion of the results are presented in the following sections.

6.2.6. Introduce and Profile Stubbed Versions

The stubbed versions of the MD5 message-digest application are created using the Automated Stubbing tool. The implementation of the stubbed versions are verified by comparing the logs from running the stubbed version to the program traces of the unmodified application. Verification of the stubbed versions demonstrated that the stubbed versions, given the same input, exercise the same program behavior as the unmodified application. The profiling results, and the discussion of the results are presented in the following sections.

6.2.7. Annotate Architectural Description

As the MD5 message-digest application does not have an architectural description, we present the computed SEC of the functional element in tables.

6.3. Results

In this section we report the results of the profiling of the MD5 message-digest application, and its various stubbed versions. Three sets of experiments are performed, resulting in a total of 660 SEC and performance measurements. Combined, the measurements are collected over the course of two weeks.

SEC measurements are given in Tables 6.1–6.5. The measurements report the median energy consumption measured using the WUP power meter, median duration, and standard deviation (SD) of the duration. Performance measurements are given in Tables B.1–B.3 in Appendix B. The measurements report the CPU utilization, Memory usage, HDD utilization, and Network usage are reported as the median over all recorded samples.

The figures in this section use abbreviations for the various versions of the application: *B* for the Baseline version, *C* for the Capture version, *B*+*R* for the Baseline+Replay version, and *R* for the Replay version.

Version	Input	SEC (J)	Duration (s)	SD (s)
Baseline	8 GB	2463.01	350.00	39.87
	32 GB	10 328.57	1400.00	111.96
	64 GB	20 008.93	2795.00	160.91
Capture	8 GB	3390.57	511.50	21.30
	32 GB	13 726.27	2036.00	74.32
	64 GB	27 417.75	4070.00	47.87
Replay	8 GB	1900.39	302.50	10.31
	32 GB	7724.12	1198.50	20.90
	64 GB	15 911.48	2389.50	36.12

Table 6.1.: Software energy consumption of experiments Ia.

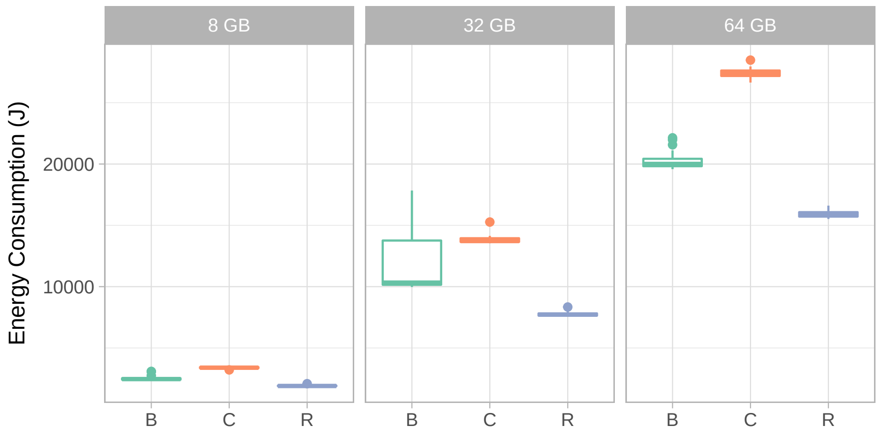


Figure 6.2.: Software energy consumption of experiments Ia.

6.3.1. Experiments Ia

The first set of experiments is performed with an input of 8 GB, 32 GB and 64 GB of data. Experiment runs are repeated 20 times for each combination of application version and input, for a total of 180 measurement samples. The SEC measurements of are summarized in Table 6.1, and illustrated in Figure 6.2. Table B.1 summarizes the collected performance data.

The first set of experiments is performed to validate the measurement approach, and to determine whether the stubbed versions created using the Automated Stubbing tool execute the capture and replay method properly. Therefore, only the Baseline, Capture, and Replay versions are included in the performance measurements. The Baseline+Replay version is not included, as it performs the replay precisely like the Replay version.

We make a single observation: we observe a large deviation of the duration of experiment executions for the Baseline version of the application.

Version	Input	SEC (J)	Duration (s)	SD (s)
Baseline	2 GB	622.89	88.00	7.13
	4 GB	1262.16	175.00	1.69
	8 GB	2581.39	349.50	10.62
	32 GB	10 853.30	1396.00	125.81
Capture	2 GB	851.02	130.00	1.78
	4 GB	1747.97	258.50	4.66
	8 GB	3616.80	512.50	25.45
	32 GB	14 375.29	2031.50	26.48
Baseline	2 GB	824.92	122.00	4.40
+Replay	4 GB	1664.17	242.00	6.73
	8 GB	3404.97	485.00	20.30
	32 GB	17 789.31	1938.50	30.30
Replay	2 GB	500.42	79.00	4.58
	4 GB	1012.06	154.00	1.66
	8 GB	2031.22	300.00	16.30
	32 GB	8285.99	1204.50	18.00

Table 6.2.: Software energy consumption of experiments Ib.

Input	SEC (J)	Ratio (%)	SEC (J/MB)
2 GB	334.54	50.79	0.1582
4 GB	639.47	51.37	0.1581
8 GB	1401.49	52.72	0.1696
32 GB	9411.15	85.72	0.2502

Table 6.3.: Software energy consumption of the MD5 message-digest application in experiments Ib.

6.3.2. Experiments Ib: External Factor

The second set of experiments is performed with an input of 2 GB, 4 GB, 8 GB and 32 GB of data. Experiment runs are repeated 10 times for each combination of application version and input, for a total of 160 measurement samples. The SEC measurements are summarized in Table 6.2, and illustrated in Figure 6.3. Table B.2 summarizes the collected performance data.

Given the SEC measurements, the SEC of the `MD5` functional element is computed using Equation 5.5. Table 6.3 shows the calculated SEC of the functional element. The Ratio column shows the SEC of the `MD5` functional element relative to the application baseline.

We make a few observations:

- We observe high CPU and HDD utilization of the Baseline+Replay application with 32 GB data as input.

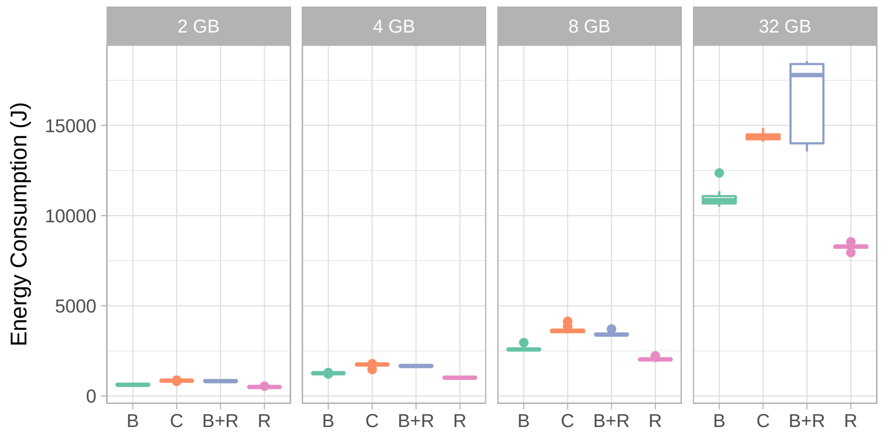


Figure 6.3.: Software energy consumption of experiments Ib.

- The SEC ratio of the MD5 functional element relative to the application baseline (Table 6.3) increases as the size of the input increases. Given an input of 8 GB the SEC ratio is 52.72 %, compared to a SEC ratio of 85.72 % given an input of 32 GB.

6.3.3. Experiments II

The third set of experiments is performed with an input of 32 MB, 64 MB, 128 MB and 256 MB of data. Experiment runs are repeated 20 times for each combination of application version and input, for a total of 320 measurement samples. The SEC measurements are summarized in Table 6.4, and illustrated in Figure 6.4. Table B.3 summarizes the collected performance data.

Given the SEC measurements, the SEC of the MD5 functional element is computed using Equation 5.5. Table 6.5 shows the calculated SEC of the functional element. The Ratio column shows the SEC of the MD5 functional element relative to the application baseline.

We make a few observations:

- The ratio of the MD5 functional element relative to the baseline varies for each input size.
- The SEC of the MD5 functional element is higher for an input of 32 MB than for an input of 64 MB.
- We observe a high CPU utilization for the Replay version of the application with 256 MB as input.

Version	Input	SEC (J)	Duration (s)	SD (s)
Baseline	32 MB	10.49	2.00	0.22
	64 MB	18.50	3.00	0.44
	128 MB	27.73	6.00	0.00
	256 MB	76.49	12.00	0.57
Capture	32 MB	19.66	4.00	0.45
	64 MB	33.03	6.00	0.00
	128 MB	61.14	10.00	0.22
	256 MB	103.91	18.00	0.88
Baseline	32 MB	21.16	4.00	0.00
+Replay	64 MB	27.61	6.00	0.37
	128 MB	56.23	9.00	0.69
	256 MB	108.38	17.00	0.37
Replay	32 MB	6.78	3.00	0.00
	64 MB	22.06	4.00	0.00
	128 MB	36.58	6.00	0.37
	256 MB	71.78	11.00	0.37

Table 6.4.: Software energy consumption of experiments II.

Input	SEC (J)	Ratio (%)	SEC (J/MB)
32 MB	12.68	119.39	0.3448
64 MB	7.47	42.02	0.1322
128 MB	20.11	59.36	0.1493
256 MB	33.98	45.59	0.1147

Table 6.5.: Software energy consumption of the MD5 message-digest application in experiments II.

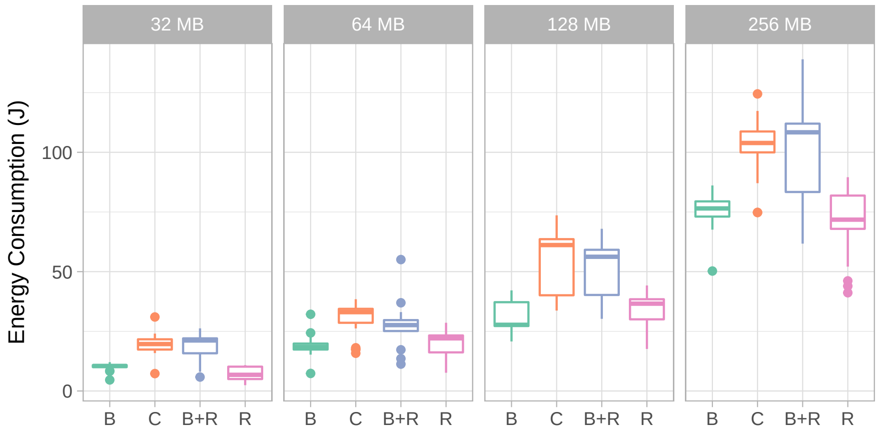


Figure 6.4.: Software energy consumption of experiments II.

6.4. Discussion

In this section we discuss the results and the findings of the software energy consumption experiments with the MD5 message-digest application, where we applied the StEP method combined with the Automated Stubbing method.

6.4.1. Duration of Experiments Ia

The SEC measurements of the first set of experiments (Table 6.1) show that for the Baseline version of the application, the SD of the duration of the experiment runs becomes increasingly large as the input size increases. Particularly, the Baseline version of the application with 64 GB of input has a SD of 160.91 s. We observe the large SDs only for the Baseline version of the application, the Capture and the Replay versions have a smaller SD. Analysis of the collected performance measurements do not indicate that the large SD can be attributed to other software processes running on the test server. Therefore, we expect that the deviation of the duration is caused by a side effect of the MD5 message-digest application. We suspect the side effect to be the .NET garbage collector.

First, the memory available to the test server is 8 GB, whereas the inputs supplied to the application do not fit completely into memory. As memory is managed automatically for C# applications, the garbage collector will allocate and free memory for the application to process the inputs.

Second, the performance data summarized in Table B.1, shows that the application is primarily memory intensive. The CPU and HDD utilization remains stable as the input to the application increases, in contrast to the memory and network usage which increase with larger inputs.

Based on the initial analysis, the second set of experiments were designed to test our hypothesis that the execution time of the experiments could be influenced by the .NET garbage collector. We expect the execution time to be more consistent for inputs smaller than 8 GB. Hence, we select two inputs smaller than 8 GB: 2 GB and 4 GB, and repeat the experiments with 8 GB and 32 GB of input.

6.4.2. Influence of the Garbage Collector in Experiments Ib

The second set of experiments were performed to test the hypothesis that the execution time of the experiments with the MD5 message-digest application are influenced by the .NET garbage collector. Table 6.2 summarizes the SEC measurements of experiments Ib.

The SEC measurements of the Baseline version with an input of 32 GB, show a large SD similar to the Baseline version with the same input from experiments Ia. The SD of

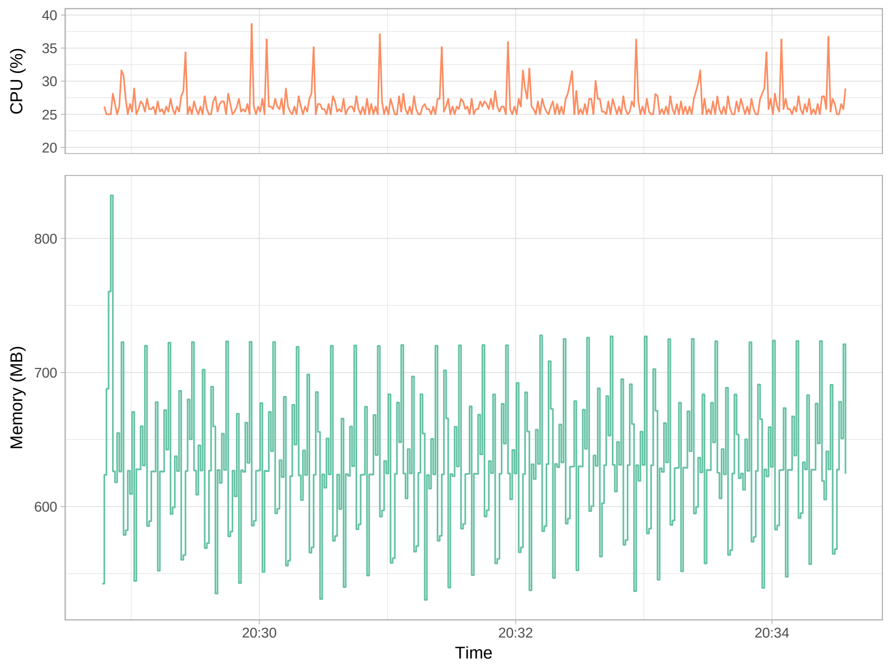


Figure 6.5.: CPU utilization and memory usage of an experiment run with an input of 8 GB.

the other versions and input sizes are smaller.

Figure 6.5 shows the CPU utilization and memory usage of a single experiment run of the application with an input of 8 GB. During the highlighted experiment run the following performance measurements were collected: CPU utilization of 26.50 %, memory consumption of 9744 MB, HDD utilization of 7.15 %, and network usage of 1.84 MB received/0.15 MB sent. The memory usage pattern of the experiment run indicates that there is a garbage collector active, as can be seen when comparing to the memory usage pattern of a generational garbage collector [32], illustrated in Figure 6.6. memory is allocated when the memory usage increases, and the garbage collector collects garbage when the memory decreases.

Comparing the CPU utilization and memory usage pattern of the single experiment run side by side, we cannot identify a direct relation. When the garbage collector is active and memory is freed we expect the CPU utilization to increase, but the Figure does not show an increase in CPU utilization consistently.

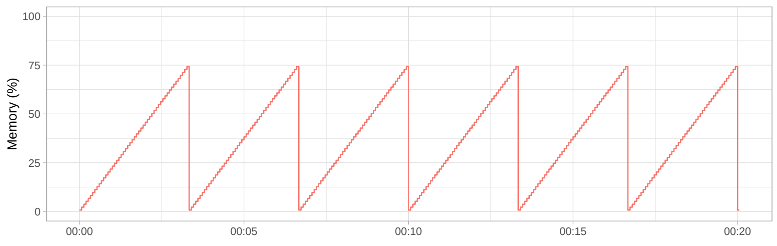


Figure 6.6.: Generational Garbage Collector memory usage pattern, after [32, Fig. 11].

6.4.3. Software Energy Consumption of the Functional Element

The stubbed versions of the MD5 message-digest application of experiments Ia (Figure 6.2), and of experiments Ib (Figure 6.3) show the expected energy consumption pattern — as discussed in Section 5.3. The SEC of the *Capture* version and the SEC of the *Baseline+Replay* are higher than the SEC of the *Baseline* version. Furthermore, the SEC of the *Replay* version is lower than the SEC of the *Baseline*.

The results of experiments Ib give an indication of the SEC of the `MD5` functional element, reported in Table 6.3. The functional element uses on average 0.1620 J/MB (SD = 0.0152) for inputs smaller than or equal to 8 GB. For larger inputs, the functional element uses on average 0.2502 J/MB (SD = 0.0705). Thus, the MD5 message-digest application uses 1.54 times more J per MB when processing 32 GB of input than when processing 2 GB to 8 GB of input.

The results of experiments II give an indication of the SEC of the `MD5` functional element as well. Table 6.5 summarizes the computed SEC of the functional element. We observe a functional element which consumes more energy than the application as a whole, the SEC of the functional element is larger than the SEC of the application given an input of 32 MB. Moreover, the functional element consumes less energy when given an input of 64 MB compared to an input of 32 MB. We expected the SEC of the functional element to increase as the input increases, however for small inputs this appears not to be the case. We suspect that the energy consumed by the `MD5` functional element is too small compared to the SEC of the instrumentation introduced by the Automated Stubbing method.

6.4.4. Performance

The collected performance measurements show that the MD5 message-digest application is primarily memory intensive. The CPU and HDD utilization remains stable as the input to the application increases, in contrast to the memory and network usage which increase as the input increases.

In the performance measurements of experiment Ib (Table B.2) we observed a high CPU and HDD utilization of the Baseline+Replay application with 32 GB data as input. Inspection of the performance logs shows that the high CPU and HDD utilization can be attributed to the `ccsvchst.exe` process. The process is part of the anti virus software on the system, and was likely performing a scan of the system or an update. Because of company policy the anti virus software cannot be disabled for the servers in the experiment environment.

In the performance measurements of experiment II (Table B.3) we observed a high CPU utilization for the Replay version of the application with 256 MB as input. Analysis of the performance logs shows that the high CPU utilization is caused by the `ccsvchst.exe` process as well, analogous to the high CPU utilization observed in the performance measurements of experiment Ib.

6.5. Reflection on Experiments

From the results in this section we can conclude that it is feasible to compute the SEC of a selected functional element using the StEP method combined with the Automated Stubbing method. There are, however, limitations which have to be taken into account when applying the approach.

First, the SEC of the selected functional element should be significantly larger than the SEC caused by the instrumentation inserted for the Automated Stubbing method. Otherwise, the SEC of the selected functional element 'disappears' in the dispersion of the SEC caused by the instrumentation — it becomes impossible to determine whether the SEC of the functional element was caused by the element itself, or whether it was caused by a deviation in the SEC measurements. The load caused by the test scenario should be large enough to negate this effect. Further research is required to learn the minimum load needed to perform valid calculations for the SEC of the functional element with the Automated Stubbing method.

Second, further research is required to understand how external factors such as the garbage collector or anti virus software influences the SEC. We have seen that the SEC of the `MD5` functional element with an input of 32 GB uses 1.54 times more J/MB compared to smaller inputs.

Chapter 7

Case Study

A case study is performed at the case company Centric to demonstrate the feasibility of using the Automated Stubbing approach to determine the SEC using the StEP method for a real-world application.

This chapter is organized as follows. First, we discuss the application under test for the case study (Section 7.1). Second, we discuss how the StEP method and the Automated Stubbing method are applied for the case study experiments (Section 7.2). Third, we show the results obtained from the execution of the experiments (Section 7.4). Fourth, we discuss the experiments, results, and findings (Section 7.5 and 7.6).

7.1. Introduction

The application under test for the case study is *Motion*[1]: a human resources and payroll application developed at the case company Centric. Motion allows an organization to maintain a database containing employees and their records; provide a platform for employees to keep their information up-to-date, track their time off work including sick days, and process expense statement forms; perform analyses to gain insights in the performance of the organization; and, automate the payroll process. The Motion application is deployed in a cloud environment, and can be accessed online or on mobile devices.

To set up the experiments with Motion, a trip was made to the Centric office in Romania to consult with the software architects, developers, and testers of the application. With their help we selected the functional element to study, we set up a local deployment of the application suitable to perform energy consumption measurements on, and designed a test scenario based on the automated tests that they use during development.

[1]`https://www.centric.eu/NL/Default/HR-software/Motion`

7.2. Applying the StEP Method

The StEP method combined with the Automated Stubbing method is applied to determine the SEC of the application under test. In this section we discuss how we perform the activities of the combined method for the case study experiments, following the process outlined in Section 5.1.

7.2.1. Select Functional Elements

The application under test is developed in C#, and deployed in a cloud environment at the case company. The application consists of three core components: a web front end, a RESTful service[2] back end, and a database.

The application back end contains the business logic of the application, and hosts the computationally expensive parts of the code of the application. Business logic is separated in modules which use an adapter interface [51] for the modules to be interchangeable. Based on the requirements that we set for the use with the Automated Stubbing tool (Section 4.1), few functional elements contain computationally expensive algorithms which could cause a high energy consumption, and for which we can determine that the side effects are limited to the selected functional element.

In consulatation with software architects and developers of the application under test, a single functional element is identified for which we want to determine the SEC. The selected functional element, *Role Assignment*, allows users of the application under test to assign user-defined roles to employees and users in the system. The core functionality of the selected functional element split into three activities:

A. *GetItems*: query which returns the collection of defined roles;
B. *SaveItems*: saving of roles assigned to a user;
C. *GetRoles*: query the roles assigned to a user.

The Role Assignment element is part of the RESTful service back end of the application.

7.2.2. Adapt Architectural Description

Each activity of the core functionality is mapped to the selected functional element in Figure 7.1. The figure shows an adapted architectural description of the application in a high-level overview to fit the core functionality of the selected functional element. Because of the size of the application the adapted architectural description only highlights some architectural elements.

[2]Web service which implements the Representational State Transfer (REST) architectural style [49, 50].

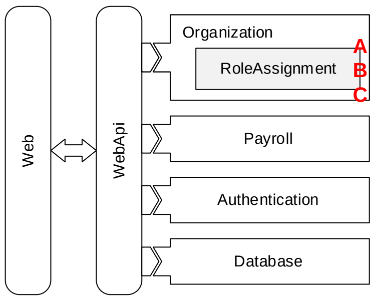


Figure 7.1.: Adapted architectural description of Motion.

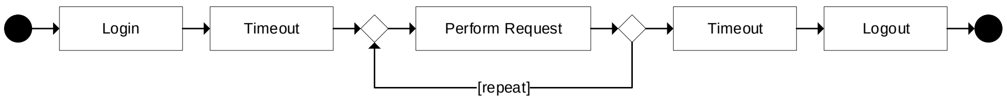


Figure 7.2.: Apache JMeter test case activities.

7.2.3. Create Test Scenario

A test case is created for each activity that is part of the core functionality of the selected functional element. Each functionality can be performed by performing a REST request to the service back end of the application. Test cases for each functionality are developed with Apache JMeter[3]. Stress of the functional element is varied by performing the REST requests 25, 100, 250 and 1000 times. We refer to the number of times the REST request is performed as the number of *users* given as input for the test case, because each REST request is a request from a user in practice. The activities performed by a test case are shown in Figure 7.2. We discuss each activity in detail.

First, HTTP requests are made to the application to perform the login procedure to obtain an access token. The access token is required to communicate with the secured application back end. After the login procedure is completed, a timeout is included to let the energy consumption of the system stabilize. The SEC measurements focus solely on the functionality exposed by the Role Assignment functional element. The login and logout procedures are not part of the functionality, and the energy consumption that they cause should not be included in the energy consumption measurements performed during the test case. A timeout of 2 minutes is determined by experimentation to be sufficient.

Second, the functional element is stressed by repeatedly performing REST requests. The requests are performed in sequence: only when a request is finished is the next request performed.

Third, a timeout is included to prevent the logout procedure from influencing the SEC

[3]`https://jmeter.apache.org/`

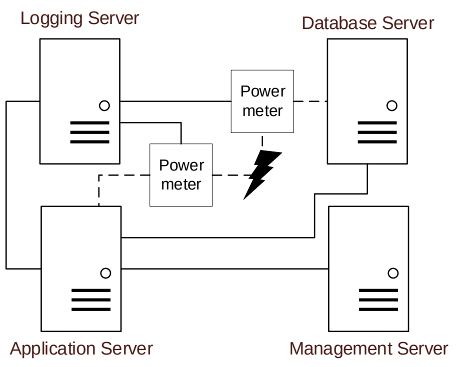


Figure 7.3.: Case study environment.

of the functional element. The logout procedure is performed to be able to repeat the activities. It is not possible to obtain an access token with the login procedure would the logout procedure not be performed.

7.2.4. Determine Measurement Approach

The Motion application is deployed on the servers in the experiment environment, as discussed in Section 3.2, to resemble the cloud environment on which the application is deployed for production. We use the measurement protocol outlined in Section 3.3, and the guidelines presented in [1, 42].

Energy Consumption and Performance Measurements

Energy consumption measurement data is gathered using hardware measurements and software measurements. The hardware measurements use the Watts UP? PRO (WUP) power meter device, the software measurements use Microsoft Joulemeter energy profiler. Microsoft Performance Monitor is used to measure the performance of the test server. The clocks of the different systems are synchronized using the Network Time Protocol, to be able to relate events that occur on different systems.

Figure 7.3 illustrates the case study experiment environment. The servers used in the environment are HP DL360 G7 servers, except the logging server, which is a HP DL380 G5. The application server hosts the web front end and the RESTful service back end, the database server hosts the Microsoft SQL database server. The logging server is used to collect the power measurements, and the management server is used to control the experiments and host the MongoDB server instance. The dashed lines indicate power cables, the lightning bolt is the power source. The solid lines are network and USB cables used to transfer data between the systems.

Software Energy Consumption

The SEC of the application is computed by subtracting the idle energy consumption from the energy consumption measured during the experiments, following from Equation 5.1. The SEC is calculated by taking the average power between two measurements multiplied by the time between measurements, and taking the sum over the duration of the measurements.

Computation of the SEC requires the idle power consumption to be known. Moreover, the Joulemeter energy profiler requires the power consumption under full load for calibration of its measurements. The HP DL360 G7 servers in the experiment environment are newly deployed by the case company for the case study experiments. Therefore, the idle power consumption and the power consumption when the servers are under full load are determined in preparation. We will discuss these measurements and their results in Section 7.3.

Measurement Protocol

The measurement protocol presented in Section 3.3 is used to perform the experiments. The case study test runs are automated by executing a batch script. Following the profiling order of the StEP method combined with the Automated Stubbing method, the Apache JMeter test case is repeated 10 times for each input and application version. Between each test run the batch script waits for the specified cool down time of 15 minutes. A limited number of repeats is selected due to the limited time available in the experiment environment. Performing a single test case for one application version takes about 12 hours to complete, including the cool down time between runs when repeated 10 times.

7.2.5. Profile Baseline

The Baseline version of the application is profiled using the created test scenario, and the determined measurement approach. The collected SEC and performance measurements are used to verify the created test scenario. The profiling results, and the discussion of the results are presented in the following sections.

7.2.6. Introduce and Profile Stubbed Versions

The stubbed versions of the application are created using the Automated Stubbing tool. The implementation of the stubbed versions are verified by comparing the logs from running the stubbed version, to the program traces of the unmodified application. Verification

Server	Idle (W)	Load (W)	Delta (W)
Application	96.7793	161.3181	64.5388
Database	78.3928	148.0000	69.6071
Management	102.9494	210.7484	107.7990

Table 7.1.: Power consumption of HP DL360 G7 servers in the experiment environment.

of the stubbed versions demonstrated that the stubbed versions, given the same input, exercise the same program behavior as the unmodified application. The profiling results, and the discussion of the results are presented in the following sections.

7.2.7. Annotate Architectural Description

After the profiling of the application is finished, the adapted architectural description is annotated with the measured SEC.

7.3. Idle and Load Energy Consumption

The idle power consumption and power consumption under full load are determined for the HP DL360 G7 servers in the experiment environment, as the servers are newly deployed for the case study experiments. The idle power consumption is used for the computation of the SEC. The idle and load power consumption are used to calibrate the Joulemeter power profiler software on the test servers. Table 7.1 presents the measured power consumption when running idle or under a full load of the servers.

To measure the power consumption when the servers are running idle, power consumption measurements are taken over a timespan of 60 hours using the WUP power meter device. The idle power consumption is taken as the mean measured power consumption.

To measure the power consumption when the servers are under full load, the HeavyLoad[4] application is used to stress the servers, following the process outlined in [14]. A total of ten runs using the HeavyLoad application are performed, where each run puts the server under a full load for one hour. The power consumption under full load is taken as the mean measured power consumption over the ten runs. The reported delta is the difference between the power consumption under full load, and the power consumption when running idle.

[4]`https://jam-software.com/heavyload/`

7.4. Results

In this section we report the results of the profiling of the Motion application, and its various stubbed versions. Four sets of experiments are performed, resulting in a total of 640 SEC and performance measurements. Out of the 640 measurement samples, one sample was removed because the application returned the HTTP status code *500 Internal Server Error*. Inspection of the logs produced by Motion revealed a serialization exception. Serialization was implemented for the required classes, and the updated application did not throw any more exceptions. Review of the event logs confirmed that the serialization was implemented correctly.

SEC measurements are summarized in Figures 7.4–7.11. Performance measurements are given in Tables C.1–C.8 in Appendix C. The measurements report the CPU utilization, Memory usage, HDD utilization, and Network usage are reported as the median over all recorded samples.

The figures in this section use abbreviations for the various versions of the applications; *B* for the Baseline version, *C* for the Capture version, *B*+*R* for the Baseline+Replay version, and *R* for the Replay version.

7.4.1. Experiments Ia

The first set of experiments is performed using the *GetItems* test case, stressing the selected functional element 25, 100, 250 and 1000 times. Figure 7.4 shows the duration of the experiment runs. Figure 7.5 illustrates the measured SEC, reported separately for the application and database servers. Tables C.1 and C.2 summarize the collected performance data.

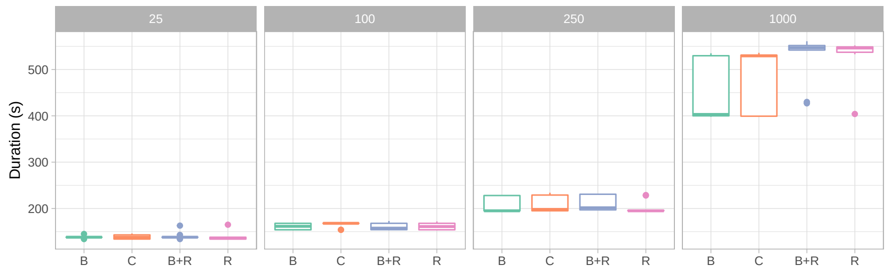


Figure 7.4.: Experiment run duration of case study Ia.

We make the following observations:

- The duration of the Baseline and Capture experiments with an input of 1000 users,

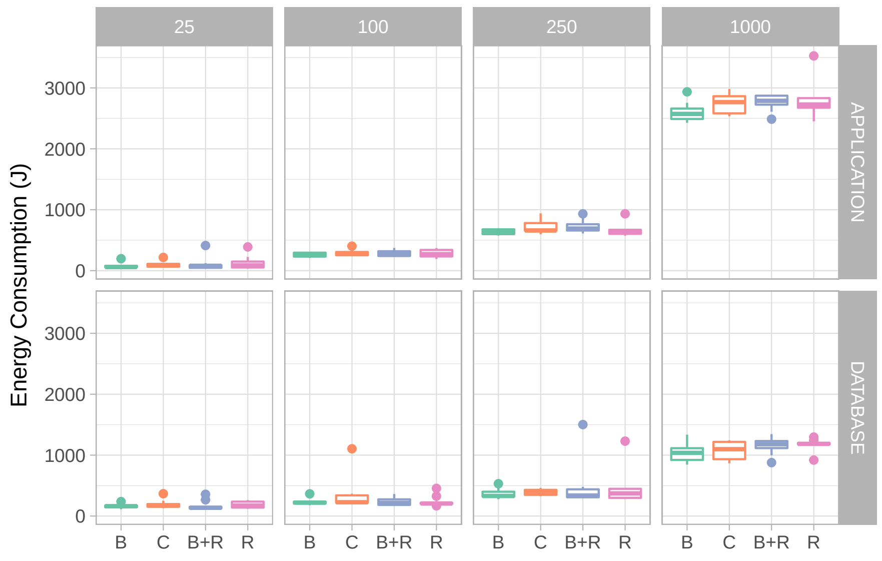


Figure 7.5.: Software energy consumption of case study Ia.

show a large spread in its distribution compared to the other versions and inputs.

- The SEC of the stubbed versions do not show the expected SEC pattern discussed in Section 5.3; the SEC of the Replay versions is equal to or larger than the SEC of the Baseline versions.

7.4.2. Experiments Ib: External Factor

The second set of experiments is performed using the *GetItems* test case, stressing the selected functional element 25, 100, 250 and 1000 times. The HP DL360 G7 servers are reconfigured to use a single network interface, in constrast to the two network interfaces they had previously. Figure 7.6 shows the duration of the experiment runs. Figure 7.7 illustrates the measured SEC, reported separately for the application and database servers. Tables C.3 and C.4 summarize the collected performance data.

We make the following observations:

- The duration of the Baseline and Capture experiments with an input of 1000 users, do not show the large spread in its distribution compared to the other versions and inputs as we observed in the first set of experiments;

- The SEC of the stubbed versions do not show the expected SEC pattern discussed in Section 5.3; the SEC of the Capture and Baseline+Replay versions is smaller than

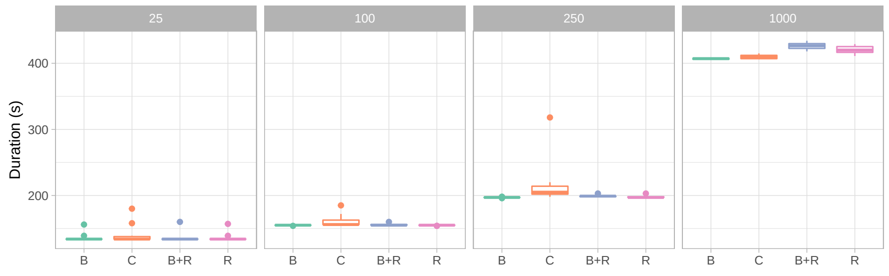


Figure 7.6.: Experiment run duration of case study Ib.

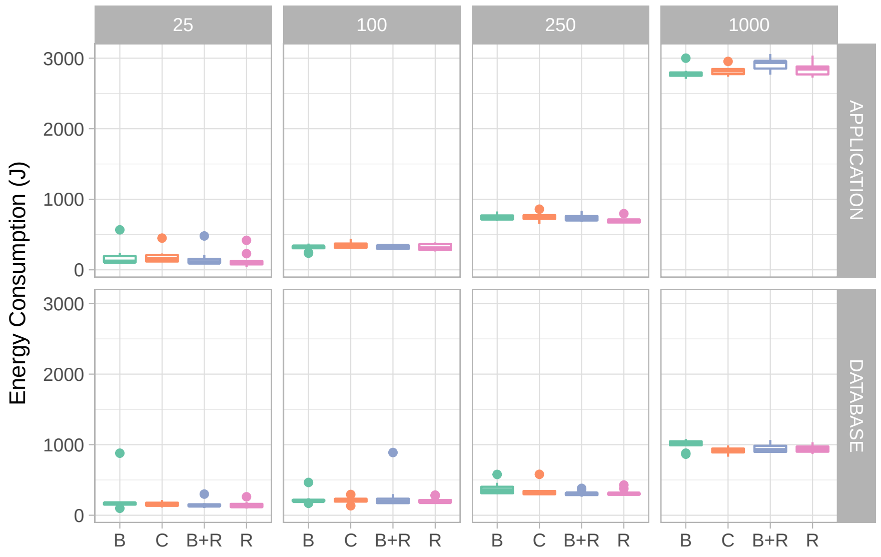


Figure 7.7.: Software energy consumption of case study Ib.

the SEC of the Baseline version.

7.4.3. Experiments II: MongoDB on the Same Server

The third set of experiments is performed using the *GetItems* test case, stressing the selected functional element 25, 100, 250 and 1000 times. The MongoDB server is moved from the management server to the application server. Figure 7.8 shows the duration of the experiment runs. Figure 7.9 illustrates the measured SEC, reported separately for the application and database servers. Tables C.5 and C.6 summarize the collected performance data.

We make the following observation: the SEC of the stubbed versions do not show the ex-

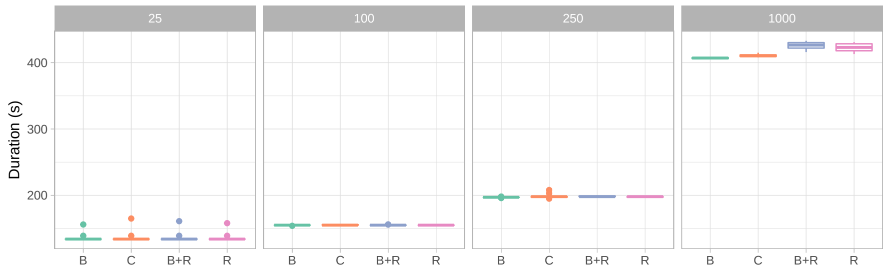


Figure 7.8.: Experiment run duration of case study II.

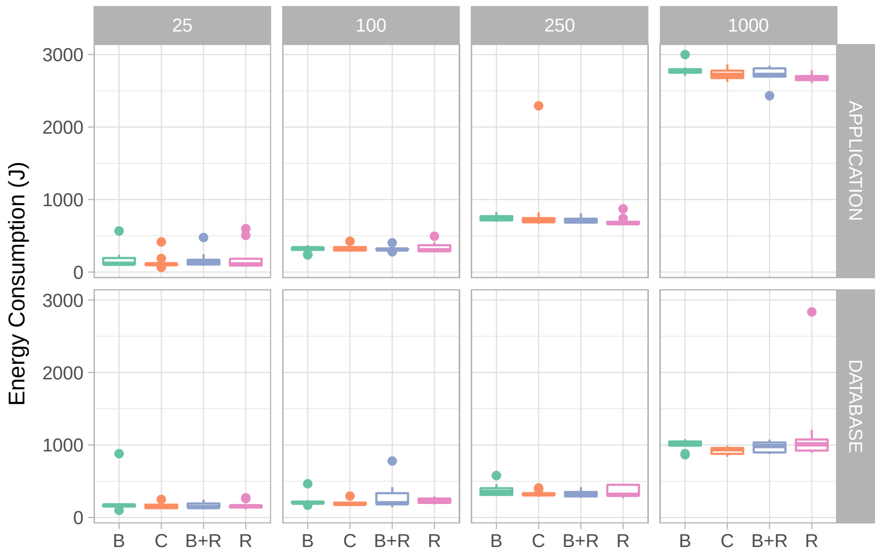


Figure 7.9.: Software energy consumption of case study II.

pected SEC pattern discussed in Section 5.3; the SEC of the Capture and Baseline+Replay versions is smaller than the SEC of the Baseline version.

7.4.4. Experiments III: A Manual Stub

The third set of experiments is performed using the *GetItems* test case, stressing the selected functional element 25, 100, 250 and 1000 times. The stubs created with the Automated Stubbing tool are replaced by a manual built stub, which returns a predetermined answer from the functional element. As the manual stub does not implement the functionality of the Automated Stubbing tool, the Capture and Baseline+Replay versions are not profiled. Figure 7.10 shows the duration of the experiment runs. Figure 7.11 illustrates the measured SEC, reported separately for the application and database servers.

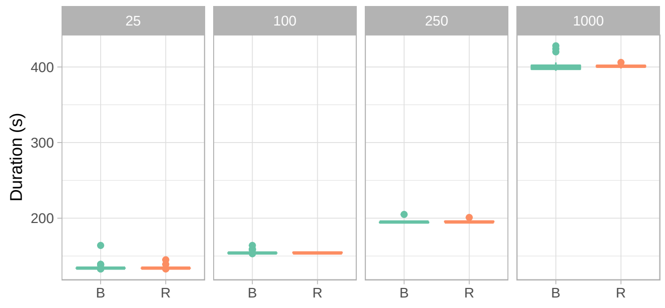


Figure 7.10.: Experiment run duration of case study III.

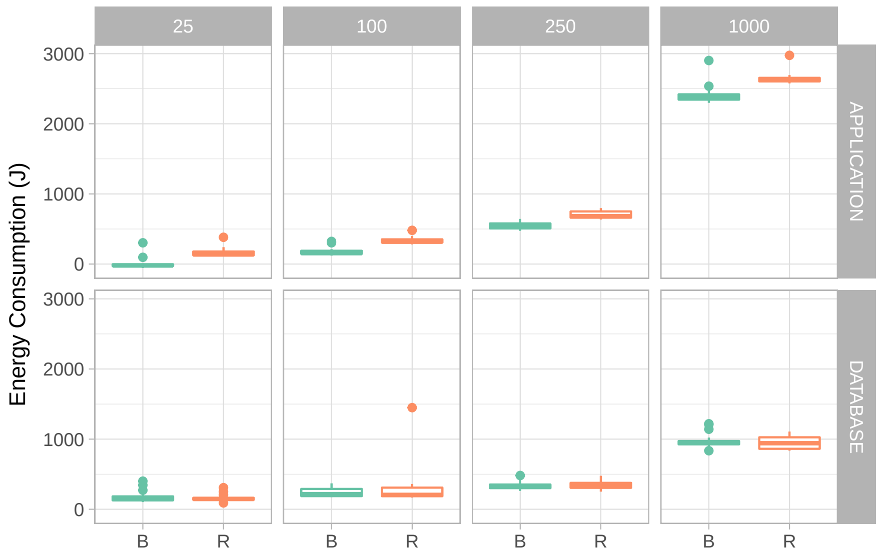


Figure 7.11.: Software energy consumption of case study III.

We make the following observation: the SEC of the stub is larger than the SEC of the baseline.

7.5. Discussion

In this section we discuss the results and observations of the software energy consumption experiments as part of the case study performed at the case company Centric with the Motion application. We applied the StEP method combined with the Automated Stubbing method to perform the case study experiments.

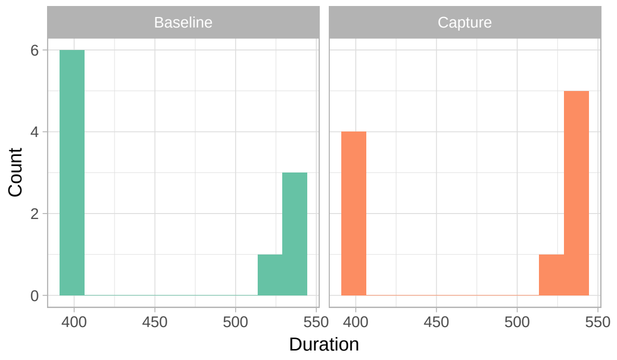


Figure 7.12.: Distribution of case study Ia duration.

7.5.1. The Impact of Two Network Interfaces

We observe a large spread in the distribution of durations of the Baseline and Capture version with an input of 1000 in the first set of experiments in Figure 7.4. If we plot the durations of the two versions in a histogram, as shown in Figure 7.12, the distribution of durations appears to be split in two. We suspect this to be caused by an external factor of the experiment environment. Because the experiments use MongoDB for the event log, which is placed on a server separate from where the application is deployed, we decided to repeat the experiments to identify the impact caused by MongoDB on the capture and replay. The experiments are repeated with MongoDB placed on the same server as the application, and the experiments are repeated with a stubbed version of the application which does not use MongoDB.

Analysis of the performance data of the test servers shows unusual usage of the network interfaces. Figure 7.13 shows a fragment of the usage of the network interfaces on the database server. The fragment shows six runs of the Baseline version of the application with an input of 1000 users. The areas highlighted in grey indicate when an experiment run is performed.

We make two observations which support our hypothesis. First, the throughput of the network interfaces differ. The throughput of *Ethernet 0* is 457.54 KB/s (SD = 3.78 KB/s) on average, whereas the throughput of *Ethernet 1* is 313.49 KB/s (SD = 1.83 KB/s) on average. Second, the duration of the experiment runs varies depending on the network interface that is active. Experiments using *Ethernet 0* have a duration of 401 s (SD = 2.58 s) on average, whereas experiments using *Ethernet 1* have a duration of 530 s (SD = 2.71 s) on average. The observations show that the duration of the experiments is tied to the throughput of the network interfaces.

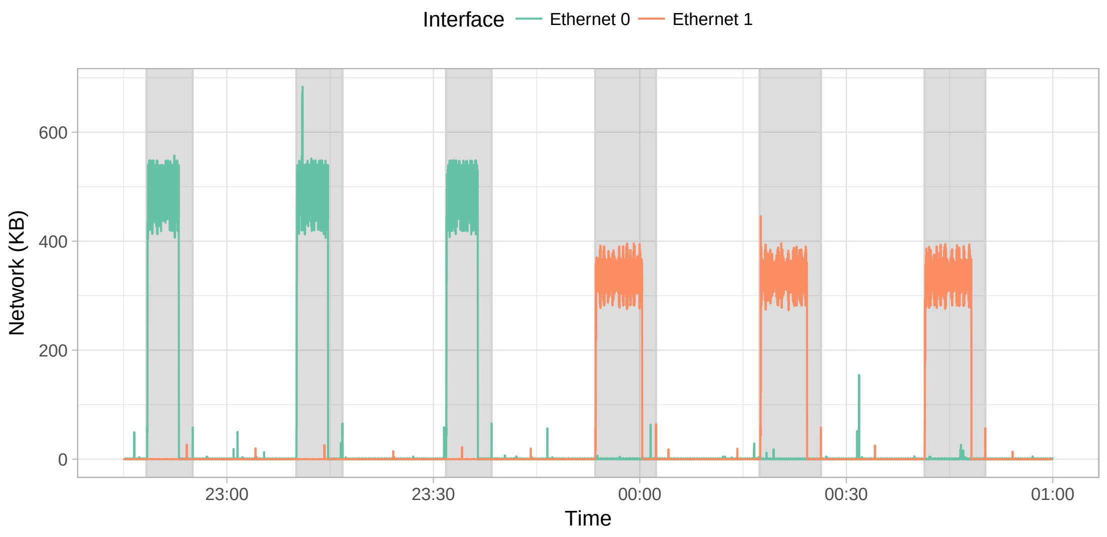


Figure 7.13.: Network usage pattern of case study Ia, Baseline version with an input of 1000 users.

7.5.2. Using a Single Network Interface

For the second set of experiments the HP DL360 G7 servers are reconfigured to use a single network interface. We think that using a single network interface eliminates the network as external factor, and improves the distribution of experiment durations observed in experiments Ia.

The durations of experiments Ib are shown in Figure 7.6. The figure shows that for the Baseline and Replay versions with an input of 1000 the dispersion of the durations is much smaller compared to experiments Ia. This confirms our expectations.

7.5.3. Software Energy Consumption of Motion

Due to the limited time remaining for the case study experiments, and the network interfaces eliminated as external factor, the decision was made to focus on a single test scenario where we vary the way in which MongoDB is used as event log. Thus, the effect on the SEC caused by MongoDB can be indentified. Case study Ib has MongoDB deployed on the management server, case study II has MongoDB deployed on the application server, and case study III does not use MongoDB. Figures 7.7, 7.9 and 7.11 show the results of the SEC measurements.

The stubbed versions of the Motion application do not show the expected SEC pattern discussed in Section 5.3: the SEC of the Capture and Baseline+Replay versions is smaller

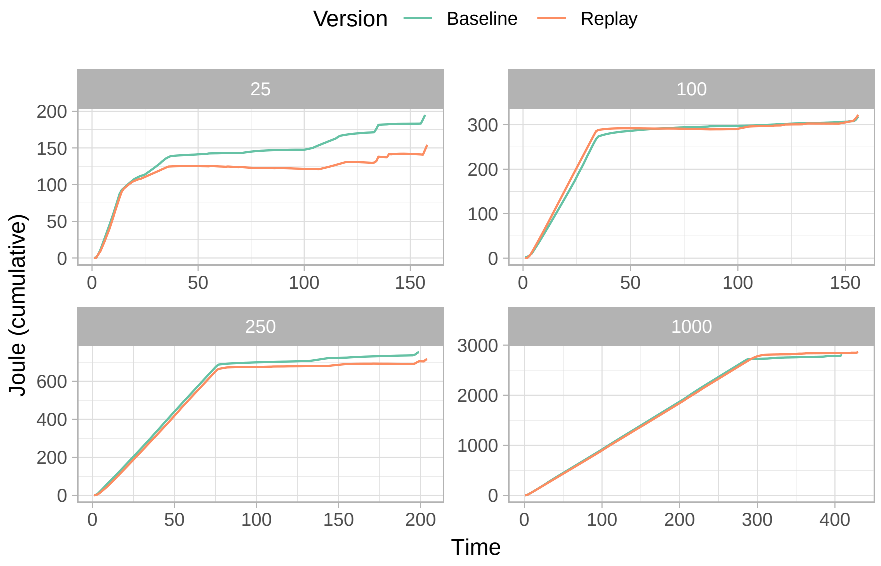


Figure 7.14.: Cumulative energy consumption of case study Ib.

than the SEC of the Baseline version (case study Ib and II); and, the SEC of the stub is larger than the SEC of the baseline (case study III). We suspect that the behavior that we see has the same cause as the SEC measurements of the preliminary experiments II. The energy consumed by the Role Assignment functional element is too small compared to the SEC of the instrumentation introduced by the Automated Stubbing method.

The cumulative SEC of the Baseline and Replay version in case study Ib are shown in Figure 7.14. SEC measurements are averaged over the 10 runs. Given an input of 25 and 250 users, the cumulative SEC of the Replay version is consistently smaller compared to the cumulative SEC of the Baseline version. But given an input of 100 and 1000 users, the cumulative SEC of the versions overlap and cross one another. Because the cumulative SEC of the Replay is not consistently smaller compared to the cumulative SEC of the Baseline, we cannot determine the SEC of the functional element.

7.5.4. Performance

The collected performance measurements show that the work exercised on the Motion application does not stress the application. The CPU utilization (max = 6.09 %), memory usage (max = 106.21 MB), and HDD utilization (max = 0.17 %) are all limited. But we observe a pattern in the network usage for all case study experiments. The majority of the network usage appears to be limited to communication between the Application server and the Database server.

We analyzed the network usage to determine whether the data sent and received from the different servers come from the same distribution. The network usage data was tested for normality, and the Mann-Whitney U test is used for the non-normally distributed data [52]. The data sent from the application server (Mdn = 4.80 MB) is significantly different from the data received by the database server (Mdn = 2.71 MB); U = 1965.5, $p < 0.05$. The data sent from the database server (Mdn = 14.02 MB) is significantly different from the data received by the application server (Mdn = 24.47 MB); U = 1148, $p < 0.05$.

7.6. Reflection on Case Study

From the results in this section we have established that the created test scenario did not stress the application sufficiently to compute the SEC of the functional element reliably. Like the experiments with the MD5 message-digest application we have learned that to apply the Automated Stubbing method, the energy consumed by the selected functional element needs to significantly differ from the energy overhead introduced by the instrumentation from the method. Further research is required to determine the right granularity between the SEC of the functional element and the SEC of the instrumentation.

Furthermore, we identified the network configuration as an external factor which influences SEC measurements. The network configuration, in particular the throughput of the network and configuration of network interfaces, has an impact on the duration of the experiments. The impact was also observed as a large dispersion in the SEC measurements.

Chapter 8

Conclusions

In this thesis, we have proposed the Automated Stubbing method for the automatic creation of stubs using the capture and replay technique, and applied it within the StEP method. We extended the StEP method activities to include the activities required to profile the various stubbed versions created with the Automated Stubbing method. We have proposed equations to compute the software energy consumption for a selected functional element given the various stubbed versions. The Automated Stubbing method was implemented, and evaluated using a series of experiments and a case study. The research in this thesis has been laid down in the main research question RQ1.

> *RQ1: How can we automatically measure the energy consumption of software architectural components within the software development process by stubbing?*

To answer the main research question, we formulated six sub questions. Combined, the answers to the sub questions provide an answer for the main research question.

> *SQ1: How can we automate the creation of stubs in the StEP method by applying the Capture and Replay method?*

We proposed the Automated Stubbing method in Chapter 4 as answer to this question. The Automated Stubbing method applies the capture and replay method to create three stubbed versions of some software application: a Capture version, a Baseline+Replay version, and a Replay version. The stubbed versions of the software application are used together with the Baseline version of the application to determine the SEC. The Automated Stubbing method is implemented in the Automated Stubbing tool. The tool, developed in C#, uses the .NET compiler platform *Roslyn* to insert the instrumentation required by the method. A series of experiments and a case study were performed to show the feasibility of using the Automated Stubbing method.

> *SQ2: How do we isolate the software architectural components to measure its energy consumption as accurately as possible?*

Energy consumed by a software architectural component is not necessarily confined to the component itself. Side effects of the component might cause energy consumption outside of the component. Within the Automated Stubbing method, the assumption is made to restrict the method to architectural components for which the side effects are limited to the state of the component. However, further research is required to extend the Automated Stubbing method to objects for which side effects happen outside of the state of the component.

> *SQ3: How do we measure the impact of instrumentation on the energy consumption?*

In Chapter 5 we proposed two equations to measure the impact of the instrumentation. Equation 5.3 calculates the overhead of the capture instrumentation by comparing the SEC of the Capture version of some application to the SEC of the Baseline version. Equation 5.4 calculates the overhead of the replay instrumentation by comparing the SEC of the Baseline+Replay version of some application to the SEC of the Baseline version. The experiments presented in Chapters 6 and 7 demonstrate that the impact of the instrumentation can be measured using the equations.

> *SQ4: How do we measure the impact of the replay on the energy consumption?*

To measure the impact of the replay on the energy consumption we introduced the Baseline+Replay version within the Automated Stubbing method. The Baseline+Replay version performs the replay on top of the Baseline version. By combining both the work of the baseline and the work of the replay, Equation 5.4 can be used to measure the impact of the replay. The experiments presented in Chapters 6 and 7 demonstrate that the impact of the replay can be measured.

> *SQ5: How do side effects of components influence the energy consumption during capture and replay?*

We have examined multiple examples of how side effects influence the energy consumption of software applications. In Chapter 6 we explored how the garbage collector could influence SEC measurements. The garbage collector has more work to perform if more memory is required by the application under test. Further research is required to identify the impact of garbage collection on energy consumption. In Chapter 7 we discovered how network utilization plays a role in SEC measurements. Two network interfaces with differing throughput caused a large dispersion in the duration of the experiment runs. By reconfiguring the servers to use a single network interface, the dispersion was minimized.

SQ6: What are effective ways to show the measured energy consumption during the software development process?

The experiments and case study were performed in a controlled environment outside of the software development processes of the case company. In Chapter 6 we determined the SEC of the MD5 message-digest application. Because of the simplicity of the application we presented the SEC of the selected functional element in a table. In the case study, Chapter 7, we were unable to determine the SEC of the selected functional element from Motion. Therefore, only the intermediate SEC measurements are presented. Further research is required to identify effective ways to apply the Automated Stubbing method, and show the measured energy consumption during the software development process.

8.1. Future Research

In this section we discuss the main limitations of the Automated Stubbing method proposed in this thesis, and identify directions for future research.

Further Automation

The Automated Stubbing method automates a single activity of the StEP method. Additional profiling activies are required when using the StEP method combined with the Automated Stubbing method. Within our experiments, the profiling activites were partially automated by batch scripts on the test servers. Analysis of the profiling results is required throughout the Automated Stubbing method to obtain reliable measurements. Therefore, using the combined method remains a time consuming effort.

Further research can reduce the effort of using the combined method. Scripts to automate the profiling activies could be generated automatically, and can be incorporated in the automated testing used for the software development process. Automated tools to detect performance issues could be used to limit the effort required to analyse profiling results, by alerting users when performance issues occur during SEC measurements.

Side Effects

Side effects of software architectural components might cause energy consumption outside of the component under study. We made the assumption within the Automated Stubbing method that side effects are limited to the internal state of the component. Further research is required to extend the Automated Stubbing method to software architectural components with side effects outside of the component under study.

Granularity of the Software Architectural Components

We discovered from our experiments and case study that there is a minimum amount of work required by the software architectural components to collect valid SEC measurements, and to reliably determine the the SEC of the component. If the work performed by the component is too small, we cannot distinguish between the dispersion of SEC measurement samples and the actual work performed. Further research is required to discover the bounds on the minimum work required.

To Conclude

To conclude, in this thesis we presented a method to automate the creation of stubs and apply it within the StEP method. We demonstrated the feasibility of the presented method by performing a series of experiments and a case study. More work is required to optimize the implementation of the tool, understand the limitations of the Automated Stubbing method, and research side effects and the external factors — such as the garbage collector and the network utilization — which could have a significant impact on software energy consumption.

Acknowledgements

First and foremost, I would like to thank my supervisors who guided me throughout this Master's Thesis research for without them this research would not have been possible, Jurriaan Hage and Jan Martijn van der Werf at Utrecht University, and Erik Jagroep and Rob van Vliet at Centric. In addition to my supervisors, this research would not have been possible without the help of many others. In particular, I would like to thank Cosmin Sontu and Mihai Mircescu from Centric Romania for their help with setting up the case study experiments; Erwin Vergeer for his support with the hardware in the experiment environment; and, Arjan van der Ent for countless valuable discussions and showing me around the Centric office in Romania.

Appendix A

Instrumentation Examples

Listing A.1: Excerpt of the Baseline version of the MD5 message-digest application.

```
class MD5
{
    public static int DigestLength { get; } = 16;
    private void Update(byte input);
    private int DoFinal(byte[] output, int outOff);

    private byte[] xBuf;
    private int xBufOff;
    private long byteCount;
    private uint H1, H2, H3, H4;
    private uint[] X = new uint[16];
    private int xOff;

    public string ComputeHash(char[] data)
    {
        byte[] result = new byte[MD5.DigestLength];
        foreach (var b in Encoding.UTF8.GetBytes(data))
        {
            Update(b);
        }
        DoFinal(result, 0);
        return BitConverter.ToString(result).Replace("-", "").ToLower();
    }
}
```

Listing A.2: Excerpt of the Capture version of the MD5 message-digest application.

```
    class MD5
    {
        public string ComputeHash(char[] data)
        {
            var _cr__parameters = new CaptureReplay.Parameter[]{new
                CaptureReplay.ReferenceParameter("data", "Char[]",
                CaptureReplay.InstanceMap.ResolveInstance(data))};
            var _cr__state = new CaptureReplay.StateMember[]{new
                CaptureReplay.StateMember("xBuf", this.xBuf), new
                CaptureReplay.StateMember("xBufOff", this.xBufOff), new
                CaptureReplay.StateMember("byteCount", this.byteCount),
                new CaptureReplay.StateMember("H1", this.H1), new
                CaptureReplay.StateMember("H2", this.H2), new
                CaptureReplay.StateMember("H3", this.H3), new
                CaptureReplay.StateMember("H4", this.H4), new
                CaptureReplay.StateMember("X", this.X), new
                CaptureReplay.StateMember("xOff", this.xOff)};
            _cr__eventLog.Capture(new
                CaptureReplay.MethodEntry(_cr__instanceId,
                "MD5.ComputeHash(Char[])", _cr__parameters), _cr__state);
            var _cr__event = new
                CaptureReplay.MethodExit(_cr__instanceId,
                "MD5.ComputeHash(Char[])", _cr__parameters);

            byte[] result = new byte[MD5.DigestLength];
            foreach (var b in Encoding.UTF8.GetBytes(data))
            {
                Update(b);
            }
            DoFinal(result, 0);
            var _cr__expression =
                BitConverter.ToString(result).Replace("-", "").ToLower();
            _cr__eventLog.Capture(_cr__event, _cr__state,
                _cr__expression);
            return _cr__expression;
        }

        private static CaptureReplay.IEventLog _cr__eventLog = new
            CaptureReplay.MongoEventLog(CaptureReplay.EventLogMode.Capture,
            "MD5.MD5");
        private CaptureReplay.InstanceId _cr__instanceId;
    }
```

Listing A.3: Excerpt of the Replay version of the MD5 message-digest application.

```
    class MD5
```

```
{
    public string ComputeHash(char[] data)
    {
        var _cr__parameters = new CaptureReplay.Parameter[]{new
            CaptureReplay.ReferenceParameter("data", "Char[]",
            CaptureReplay.InstanceMap.ResolveInstance(data))};
        var _cr__event = _cr__eventLog.Replay(_cr__instanceId,
            "MD5.ComputeHash(Char[])", _cr__parameters);
        xBuf = (byte[])(_cr__event.State["xBuf"]);
        xBufOff = (int)(_cr__event.State["xBufOff"]);
        byteCount = (long)(_cr__event.State["byteCount"]);
        H1 = (uint)(_cr__event.State["H1"]);
        H2 = (uint)(_cr__event.State["H2"]);
        H3 = (uint)(_cr__event.State["H3"]);
        H4 = (uint)(_cr__event.State["H4"]);
        X = (uint[])(_cr__event.State["X"]);
        xOff = (int)(_cr__event.State["xOff"]);
        return (string)((_cr__event as
            CaptureReplay.MethodExit).ReturnValue ?? default(string));
    }

    private static CaptureReplay.IEventLog _cr__eventLog = new
        CaptureReplay.MongoEventLog(CaptureReplay.EventLogMode.Capture,
        "MD5.MD5");
    private CaptureReplay.InstanceId _cr__instanceId;
}
```

Listing A.4: Excerpt of the Baseline+Replay version of the MD5 message-digest application.

```
class MD5
{
    public string ComputeHash(char[] data)
    {
        var _cr__parameters = new CaptureReplay.Parameter[]{new
            CaptureReplay.ReferenceParameter("data", "Char[]",
            CaptureReplay.InstanceMap.ResolveInstance(data))};
        byte[] result = new byte[MD5.DigestLength];
        foreach (var b in Encoding.UTF8.GetBytes(data))
        {
            Update(b);
        }

        DoFinal(result, 0);
        var _cr__event = _cr__eventLog.Replay(_cr__instanceId,
                "MD5.ComputeHash(Char[])", _cr__parameters);
            xBuf = (byte[])(_cr__event.State["xBuf"]);
            xBufOff = (int)(_cr__event.State["xBufOff"]);
            byteCount = (long)(_cr__event.State["byteCount"]);
            H1 = (uint)(_cr__event.State["H1"]);
            H2 = (uint)(_cr__event.State["H2"]);
            H3 = (uint)(_cr__event.State["H3"]);
            H4 = (uint)(_cr__event.State["H4"]);
            X = (uint[])(_cr__event.State["X"]);
            xOff = (int)(_cr__event.State["xOff"]);
            return BitConverter.ToString(result).Replace("-",
                "").ToLower();
        }

        private static CaptureReplay.IEventLog _cr__eventLog = new
            CaptureReplay.MongoEventLog(CaptureReplay.EventLogMode.Capture,
            "MD5.MD5");
        private CaptureReplay.InstanceId _cr__instanceId;
    }
```

Appendix B

Experiment Results

Version	Input	CPU (%)	Memory (MB)	HDD (%)	Network Received (MB)	Network Sent (MB)
Baseline	8 GB	26.56	9878.05	4.18	1.89	0.15
	32 GB	26.56	38 810.27	3.25	7.28	0.61
	64 GB	26.56	76 957.33	3.43	14.71	1.27
Capture	8 GB	26.17	10 324.11	4.44	2.94	0.75
	32 GB	26.17	40 429.09	2.96	11.59	3.04
	64 GB	26.17	80 287.42	3.13	22.29	6.08
Replay	8 GB	26.17	2790.39	4.03	1.92	0.23
	32 GB	26.17	10 564.02	2.97	8.68	0.92
	64 GB	26.17	20 890.01	3.37	14.87	1.81

Table B.1.: Performance measurements of experiments Ia.

Version	Input	CPU (%)	Memory (MB)	HDD (%)	Network Received (MB)	Network Sent (MB)
Baseline	2 GB	26.20	2580.18	7.12	0.50	0.04
	4 GB	26.35	4983.62	7.37	0.94	0.07
	8 GB	26.53	9757.76	6.93	1.84	0.15
	32 GB	26.71	38 532.25	7.75	7.65	0.63
Capture	2 GB	25.97	2727.51	6.86	0.96	0.19
	4 GB	25.97	5351.28	6.92	1.46	0.38
	8 GB	26.15	10 286.43	6.98	2.97	0.76
	32 GB	26.33	40 343.57	7.17	12.52	3.06
Baseline	2 GB	26.08	2584.03	7.84	0.73	0.08
+Replay	4 GB	26.00	5058.30	7.07	1.45	0.15
	8 GB	26.02	9850.69	6.97	3.04	0.33
	32 GB	36.65	39 294.98	21.96	11.69	1.27
Replay	2 GB	26.26	914.74	7.05	0.57	0.06
	4 GB	26.22	1548.50	7.16	0.99	0.11
	8 GB	26.13	2643.91	6.68	1.86	0.22
	32 GB	26.52	10 935.88	6.86	7.62	0.90

Table B.2.: Performance measurements of experiments Ib.

Version	Input	CPU (%)	Memory (MB)	HDD (%)	Network Received (MB)	Network Sent (MB)
Baseline	32 MB	18.85	49.47	7.93	0.01	0.00
	64 MB	22.17	142.57	10.03	0.02	0.00
	128 MB	23.17	290.66	10.46	0.03	0.00
	256 MB	25.52	415.58	9.14	0.06	0.01
Capture	32 MB	23.24	68.63	8.29	0.03	0.01
	64 MB	23.99	175.20	8.76	0.04	0.01
	128 MB	24.70	304.66	9.86	0.06	0.02
	256 MB	25.92	515.64	8.31	0.11	0.03
Baseline	32 MB	20.99	78.68	8.60	0.03	0.00
+Replay	64 MB	24.67	163.05	9.04	0.04	0.01
	128 MB	24.66	316.37	8.65	0.06	0.01
	256 MB	25.83	495.40	9.55	0.11	0.01
Replay	32 MB	20.95	57.91	6.25	0.02	0.00
	64 MB	24.72	59.92	7.98	0.03	0.00
	128 MB	25.11	227.43	8.37	0.04	0.01
	256 MB	32.63	338.40	10.98	0.07	0.01

Table B.3.: Performance measurements of experiments II.

Appendix C

Case Study Results

Version	Users	CPU (%)	Memory (MB)	HDD (%)	Network Received (MB)	Network Sent (MB)
Baseline	25	0.84	34.86	0.09	4.46	0.97
	100	1.70	24.01	0.09	14.52	2.90
	250	2.90	37.31	0.09	34.72	6.77
	1000	5.31	68.95	0.09	135.59	25.80
Capture	25	0.97	31.10	0.11	4.48	0.99
	100	1.73	36.07	0.10	14.58	3.10
	250	3.07	41.51	0.09	34.80	7.07
	1000	4.37	97.74	0.09	136.14	28.47
Baseline	25	0.84	37.42	0.09	4.49	0.95
+Replay	100	1.77	32.60	0.08	14.63	2.88
	250	3.03	51.20	0.08	34.97	6.80
	1000	4.19	106.21	0.08	136.77	27.58
Replay	25	0.96	39.76	0.09	4.51	0.92
	100	1.79	34.36	0.08	14.62	2.84
	250	2.99	43.53	0.08	34.77	6.48
	1000	4.09	104.03	0.09	136.17	26.43

Table C.1.: Performance measurements of case study Ia of the Application server.

Version	Users	CPU (%)	Memory (MB)	HDD (%)	Network Received (MB)	Network Sent (MB)
Baseline	25	0.40	18.05	0.10	0.82	4.16
	100	0.56	19.45	0.09	2.67	14.03
	250	0.81	27.18	0.09	6.24	33.78
	1000	1.36	65.25	0.09	24.37	132.59
Capture	25	0.44	18.28	0.10	0.80	4.14
	100	0.57	24.35	0.10	2.75	14.00
	250	0.81	27.71	0.11	6.23	33.77
	1000	1.25	72.60	0.09	25.94	132.66
Baseline	25	0.37	15.98	0.07	0.81	4.11
+Replay	100	0.59	22.65	0.07	2.59	14.01
	250	0.83	28.42	0.07	6.41	33.78
	1000	1.19	80.04	0.08	25.93	132.70
Replay	25	0.55	22.21	0.12	0.74	4.14
	100	0.61	18.28	0.09	2.55	13.96
	250	1.01	27.15	0.08	5.95	33.60
	1000	1.22	85.37	0.09	24.77	131.91

Table C.2.: Performance measurements of case study Ia of the Database server.

Version	Users	CPU (%)	Memory (MB)	HDD (%)	Network Received (MB)	Network Sent (MB)
Baseline	25	1.43	42.79	0.14	4.53	1.25
	100	2.36	41.63	0.14	14.64	3.22
	250	3.45	54.12	0.11	34.89	7.58
	1000	5.51	98.25	0.11	135.98	26.48
Capture	25	1.35	36.62	0.11	4.28	0.91
	100	2.23	30.13	0.11	14.34	2.84
	250	3.39	47.55	0.15	34.49	6.68
	1000	5.89	87.03	0.17	135.11	25.67
Baseline +Replay	25	1.44	37.74	0.10	4.29	0.91
	100	2.31	33.25	0.12	14.34	2.83
	250	3.53	39.97	0.09	34.49	6.67
	1000	6.09	79.81	0.09	135.14	25.70
Replay	25	1.39	38.36	0.09	4.31	0.89
	100	2.34	31.79	0.09	14.32	2.71
	250	3.55	36.29	0.09	34.34	6.38
	1000	6.05	76.99	0.09	134.45	24.51

Table C.3.: Performance measurements of case study Ib of the Application server.

Version	Users	CPU (%)	Memory (MB)	HDD (%)	Network Received (MB)	Network Sent (MB)
Baseline	25	0.37	15.70	0.11	0.76	3.96
	100	0.59	25.39	0.10	2.57	13.50
	250	1.04	35.15	0.08	0.04	0.09
	1000	1.25	69.80	0.10	12.17	64.11
Capture	25	0.44	15.64	0.09	0.77	4.01
	100	0.65	18.47	0.08	2.59	13.57
	250	0.87	22.43	0.09	6.21	32.62
	1000	1.31	45.85	0.09	24.30	128.03
Baseline +Replay	25	0.34	14.36	0.10	0.76	3.96
	100	0.54	18.09	0.11	2.57	13.51
	250	0.76	21.61	0.08	6.21	32.60
	1000	1.18	49.92	0.09	24.32	128.11
Replay	25	0.37	15.63	0.08	0.74	4.00
	100	0.52	15.00	0.07	2.46	13.47
	250	0.80	23.52	0.07	5.91	32.42
	1000	1.19	47.41	0.09	23.14	127.34

Table C.4.: Performance measurements of case study Ib of the Database server.

Version	Users	CPU (%)	Memory (MB)	HDD (%)	Network Received (MB)	Network Sent (MB)
Baseline	25	1.32	34.82	0.11	4.33	0.91
	100	2.29	28.31	0.09	14.37	2.83
	250	3.53	39.46	0.09	34.50	6.68
	1000	5.95	82.83	0.09	135.17	25.66
Capture	25	1.31	32.55	0.08	4.29	0.94
	100	2.27	35.42	0.09	14.39	2.95
	250	3.49	38.83	0.10	34.58	6.96
	1000	6.06	101.09	0.09	135.49	26.89
Baseline	25	1.40	34.38	0.13	4.38	0.94
+Replay	100	2.28	26.50	0.14	14.49	2.87
	250	3.54	40.28	0.11	34.74	6.76
	1000	5.77	83.99	0.11	136.11	26.04
Replay	25	1.44	30.86	0.16	4.33	0.89
	100	2.30	39.85	0.10	14.41	2.75
	250	3.46	40.86	0.09	34.55	6.47
	1000	5.78	76.93	0.09	135.32	24.87

Table C.5.: Performance measurements of case study II of the Application server.

Version	Users	CPU (%)	Memory (MB)	HDD (%)	Network Received (MB)	Network Sent (MB)
Baseline	25	0.52	18.21	0.10	0.77	4.03
	100	0.67	19.10	0.10	2.58	13.51
	250	0.91	24.80	0.08	6.22	32.63
	1000	1.49	57.70	0.09	24.31	128.10
Capture	25	0.39	16.34	0.07	0.76	3.96
	100	0.55	18.00	0.08	2.58	13.53
	250	0.82	22.76	0.07	6.21	32.66
	1000	1.25	47.93	0.10	24.35	128.11
Baseline	25	0.44	19.77	0.09	0.77	4.00
+Replay	100	0.56	18.04	0.13	2.58	13.53
	250	0.79	21.84	0.09	6.22	32.66
	1000	1.29	59.67	0.09	24.30	128.07
Replay	25	0.46	17.23	0.09	0.73	3.96
	100	0.65	22.52	0.10	2.46	13.48
	250	0.90	30.17	0.10	5.93	32.49
	1000	1.52	66.67	0.09	23.15	127.42

Table C.6.: Performance measurements of case study II of the Database server.

Version	Users	CPU (%)	Memory (MB)	HDD (%)	Network Received (MB)	Network Sent (MB)
Baseline	25	0.91	20.44	0.10	4.34	0.91
	100	1.77	27.14	0.09	14.39	2.83
	250	3.00	36.28	0.08	34.54	6.68
	1000	5.31	76.87	0.10	135.18	25.66
Replay	25	1.39	25.59	0.11	4.29	0.87
	100	2.23	28.78	0.09	14.28	2.71
	250	3.44	34.16	0.08	34.30	6.37
	1000	5.89	80.52	0.12	134.42	24.51

Table C.7.: Performance measurements of case study III of the Application server.

Version	Users	CPU (%)	Memory (MB)	HDD (%)	Network Received (MB)	Network Sent (MB)
Baseline	25	0.39	16.93	0.09	0.77	4.00
	100	0.62	19.75	0.08	2.59	13.55
	250	0.81	25.68	0.08	6.22	32.66
	1000	1.29	55.84	0.11	24.31	128.17
Replay	25	0.45	17.17	0.10	0.74	4.01
	100	0.61	20.45	0.09	2.46	13.46
	250	0.88	25.03	0.08	5.92	32.45
	1000	1.36	51.32	0.11	23.16	127.41

Table C.8.: Performance measurements of case study III of the Database server.

References

[1] E. Jagroep, A. van der Ent, J. M. E. M. van der Werf, J. Hage, L. Blom, R. van Vliet, and S. Brinkkemper, "The Hunt for the Guzzler : Architecture-based Energy Profiling using Stubs," *Information and Software Technology*, 2017.

[2] J. L. Sawin, J. Rutovitz, and F. Sverrisson, *Renewables 2018 Global Status Report*. Renewable Energy Policy Network for the 21st Century (REN21), 2018.

[3] J. Beckett and R. Bradfield, "Power Efficiency Comparison of Enterprise-Class Blade Servers and Enclosures," *A Dell Technical White Paper*, 2011.

[4] E. Jagroep, "Green Software Products," PhD thesis, Utrecht University, 2017.

[5] S. Murugesan, "Harnessing Green It: Principles and Practices," *IT professional*, vol. 10, no. 1, 2008.

[6] P. Lago, S. A. Koçak, I. Crnkovic, and B. Penzenstadler, "Framing sustainability as a property of software quality," *Communications of the ACM*, vol. 58, no. 10, Sep. 2015.

[7] P. Lago, "Challenges and Opportunities for Sustainable Software," *Proceedings - 5th International Workshop on Product Line Approaches in Software Engineering, PLEASE 2015*, 2015.

[8] S. Naumann, M. Dick, E. Kern, and T. Johann, "The GREENSOFT Model: A reference model for green and sustainable software and its engineering," *Sustainable Computing: Informatics and Systems*, vol. 1, no. 4, 2011.

[9] Y. Sun, Y. Zhao, Y. Song, Y. Yang, H. Fang, H. Zang, Y. Li, and Y. Gao, "Green challenges to system software in data centers," *Frontiers of Computer Science in China*, vol. 5, no. 3, 2011.

[10] M. Dayarathna, Y. Wen, and R. Fan, "Data Center Energy Consumption Modeling: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, 2016.

[11] M. Poess and R. O. Nambiar, “Energy cost, the key challenge of today’s data centers,” *Proceedings of the VLDB Endowment*, vol. 1, no. 2, Aug. 2008.

[12] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, “A Preliminary Study of the Impact of Software Engineering on GreenIT,” 2012.

[13] A. Noureddine, R. Rouvoy, and L. Seinturier, “A review of energy measurement approaches,” vol. 47, no. 3, 2013.

[14] E. Jagroep, J. M. E. M. van der Werf, S. Jansen, M. Ferreira, and J. Visser, “Profiling energy profilers,” *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, vol. 2, 2015.

[15] N. Rozanski and E. Woods, *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2011.

[16] E. Jagroep, J. M. E. M. van der Werf, S. Brinkkemper, L. Blom, and R. van Vliet, “Extending software architecture views with an energy consumption perspective,” *Computing*, vol. 99, no. 6, 2017.

[17] ISO/IEC, *ISO/IEC 25010:2011*, 2011. [Online]. Available: `https://www.iso.org/standard/35733.html` (visited on 02/09/2018).

[18] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Third Ed. 2013.

[19] G. Procaccianti, P. Lago, and G. A. Lewis, “A catalogue of green architectural tactics for the cloud,” in *Proceedings - 2014 IEEE 8th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems, MESOCA 2014*, vol. 52412, 2014.

[20] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE Transactions on Software Engineering*, vol. 41, no. 5, 2015.

[21] T. Xie, N. Tillmann, and P. Lakshman, “Advances in unit testing,” in *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE ’16*, New York, New York, USA: ACM Press, 2016.

[22] J. Whaley, M. C. Martin, and M. S. Lam, “Automatic extraction of object-oriented component interfaces,” *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, 2002.

[23] X. Xiao, S. Thummalapenta, and T. Xie, *Advances on Improving Automation in Developer Testing*. 2012, vol. 85.

[24] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, “Automatic test factoring for java,” *Automated Software Engineering*, 2005.

[25] H. Jaygarl, S. Kim, and C. K. Chang, “OCAT : Object Capture based Automated Testing Categories and Subject Descriptors,”

[26] S. Thummalapenta, J. D. Halleux, N. Tillmann, and S. Wadsworth, “DyGen : Automatic Generation of High-Coverage Tests via Mining Gigabytes of Dynamic Traces,” in *International Conference on Tests and Proofs*, Springer, 2010.

[27] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, Third Ed, 3. 2011, vol. 1.

[28] A. Orso and B. Kennedy, “Selective capture and replay of program executions,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, 2005.

[29] *C# 6 language specification (draft)*. [Online]. Available: `https://github.com/dotnet/csharplang` (visited on 03/12/2018).

[30] ECMA International, *Standard ECMA-334: C# Language Specification*, 5th Ed. 2017.

[31] Microsoft, *Fundamentals of Garbage Collection*. [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals` (visited on 06/14/2018).

[32] P. R. Wilson, “Uniprocessor garbage collection techniques,” in *Memory Management*, Springer Berlin Heidelberg, 1992.

[33] Microsoft, *Overview of the .NET Framework*. [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview` (visited on 05/02/2018).

[34] Microsoft, *Common Language Runtime (CLR) overview*. [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/standard/clr` (visited on 05/02/2018).

[35] T. Thai and H. Lam, *.NET Framework Essentials*. O’Reilly Media, Inc., 2003.

[36] ECMA International, *Standard ECMA-335: Common Language Infrastructure (CLI)*, 6th Ed. 2012.

[37] Microsoft, *The .NET Compiler Platform ("Roslyn")*. [Online]. Available: `https://github.com/dotnet/roslyn` (visited on 02/22/2018).

[38] Microsoft, *MSBuild*. [Online]. Available: `https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild` (visited on 05/03/2018).

[39] B. Albahari and J. Albahari, *C# 6.0 in a Nutshell: The Definitive Reference*, 6th Ed. O’Reilly Media, Inc., 2015.

[40] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd Ed. Pearson Education, 2006.

[41] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. 2012, vol. 9783642290442.

[42] E. A. Jagroep, J. M. van der Werf, S. Brinkkemper, G. Procaccianti, P. Lago, L. Blom, and R. van Vliet, "Software Energy Profiling: Comparing Releases of a Software Product," in *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, New York, New York, USA: ACM Press, 2016.

[43] *MongoDB for GIANT Ideas | MongoDB*. [Online]. Available: `https://www.mongodb.com/` (visited on 04/20/2018).

[44] C.-K. Luk, B. C. Ed, F. C. G. Hi, E. D. Q. Rs, A. Tu, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI '05*, vol. 40, no. 6, 2005.

[45] N. Nethercote, "Dynamic binary analysis and instrumentation," PhD thesis, University of Cambridge, Nov. 2004.

[46] E. Jagroep, J. M. E. M. van der Werf, J. Broekman, L. Blom, R. van Vliet, and S. Brinkkemper, "A Resource Utilization Score for Software Energy Consumption," *Proceedings of ICT for Sustainability 2016*, no. Ict4s, 2016.

[47] A. E. Trefethen and J. Thiyagalingam, "Energy-aware software: Challenges, opportunities and strategies," *Journal of Computational Science*, vol. 4, no. 6, 2013.

[48] R. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321, Apr. 1992.

[49] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," PhD thesis, 2000.

[50] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, Inc., 2008.

[51] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series, 1995.

[52] A. Field, *Discovering statistics using IBM SPSS statistics*. SAGE, 2013.