Master Thesis
Master of Computing Science

ECAlogic Analysis

A Case Study for Energy-Consumption of DNS Server Implementations

Author:
Enniër Kelly
4146999

Supervised by:
prof. dr. Marko van Eekelen
dr. Rody Kersten

Second reader:
prof. dr. ir. Joost Visser

A thesis submitted in fulfillment of the requirements for the degree of Master of Computing Science in the

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

September 7, 2015

Abstract

In recent years there has been a growing interest in reducing energy consumption of software, all the while making sure they remain performant. One scenario were these two are important, and can have a great effect, is in the critical component of the Internet, the Domain Name System (DNS). However, developers of DNS server implementations have no way to determine the amount of energy used by their implementation.

This research aims to use ECALOGIC to model the energy consumption for the DNS implementations known as BIND and Unbound. ECALOGIC is a recently developed tool that aims to statically provide upper bounds on the energy consumption of software. The consumed energy is determined by giving ECALOGIC a model of the hardware and a software implementation. This thesis researches which hardware components must be modeled and develops simplified versions of the servers in ECALOGIC. These are then populated with energy consumption values. Time is spent to research how the energy consumption values are acquired. A measurement methodology is presented on how these values can be obtained. Consumption values are obtained, analyzed and used in the analysis for the DNS servers.

The analysis on the DNS servers not only aims to provide an answer on which implementation is the most energy efficient, but also look at the limitations and possibilities of ECALOGIC. This result of this analysis is than compared to another independent research that also looked at the energy consumption of DNS servers. This is followed by discussing the measurement methodology and the experience of using ECALOGIC. Improvements on the measurement method are suggested and future enhancements to ECALOGIC are presented.

Acknowledgements

This thesis presents a partial fulfillment of the requirements for obtaining the degree of Master of Science in Computing Science at the Radboud University Nijmegen, the Netherlands. It took a little longer than expected, but this thesis would have not been possible without the advice and support of many people. I would like to take this moment to thank all of them.

First and foremost, I would like to thank Marko van Eekelen and Rody Kersten for supervising me during this thesis, giving me the opportunity to work on this project and certainly their patience. Their words of encouragement kept me going in times when I doubted myself and could not see a way on how to finish this thesis. Every meeting would instill a new sense of determination to continue working on this research. I would also like to thank Joost Visser for agreeing to be the second reader for this thesis.

I would also like to thank Bo Merkus and Erik Hoestra of `SEFlab` for giving me all the time I needed to use their energy consumption measurement setup, their help when the hardware failed to work and the scripts needed to convert the measured raw data into readable power consumption values.

I would also like to send my thanks Matthijs Mekking of `NLnet Labs` and Jelte Jansen of `SIDN labs` for helping me to find my way in the source code of `Unbound` and `BIND`. Their guidance certainly steered me in the right direction in such large code bases.

I would like to express my gratitude to my family for their support and the constant reminders that I am working on a thesis that needs to be finished and handed. I would like to thank my brother for his patience when I did not have the time to help as much as I would like with our company.

Last but least, I would like to thank my girlfriend for her continuous support, encouragement and, more importantly, always believing in me.

Thank you all,

Enniër Kelly
September 2015

Contents

Chapter 1

Introduction

1.1 Introduction

In recent years there has been a growing interest in reducing energy-consumption of software. The challenge lies in delivering performant software while reducing its energy-consumption. One example where energy-consumption reduction can have great effect is in a critical component of the Internet namely, the Domain Name System (DNS). The service that the DNS provides is that of translating user-friendly domain names to Internet Protocol (IP) addresses. Each visit to a website requires the use of DNS server software. Google alone performed 5.7 billion searches in 2014[1] per day, and 5.9 billion in 2013, that each required the services provided by DNS. Therefore, knowing the energy-consumption of DNS server software is certainly of interest. More so, because the ICT sector accounts for approximately 2% of global carbon emissions[2]. Thus, making DNS servers more energy-efficient can help reduce global carbon emissions. However, developers of DNS server implementations, or of software in general, currently have no simple way to ascertain how much energy their applications consume.

Reducing energy consumption of software is not only important for DNS servers, the backbone of The Internet, but also for servers used for large scale systems and devices found in the consumer market. Take for example companies such as Google, Yahoo, Microsoft and Amazon that have to support large data centers containing thousands of machines that provide the needed processing capabilities to support their business services [2]. Furthermore, trends in the consumer technology market show that more and more consumers are moving towards mobile consumer oriented devices such as smart phones and tablets. In their 'Gartner Says' series, Garnter reports in [10] that in 2014 alone more than 1.2 billion units of smart phone devices had been sold worldwide. Essentially making smart phone sales two-thirds of the total global phone market in 2014. However, no matter how powerful these mobile devices have all become during the last few years, they are still limited in the amount of energy they can retain [16]. Therefore, making applications for mobile devices more energy efficient is of utmost importance in extending the battery life of these devices and, at the same time, enhance the user experience.

Currently, one way of determining the energy-consumption footprint of applications is to take measurements while it is operational. However, taking measurements is tedious work and requires measurement equipment that may not be readily accessible to everyone. Moreover, energy consumption is only measured for a finite set of test inputs. Generic results, valid for *any* input, cannot be achieved using this approach. One laboratory in the Netherlands that is equipped to take such measurements is the Software Energy Footprint Lab (SEFLab), located in Amsterdam. The laboratory is in possession of a number of different production grade servers. Each server is equipped to take accurate measurements directly from most of its hardware components. This is done by equipping the servers with power sensors on the power lines. The measurement setup is seen as a black box that executes the software for a determined period of time on the server and outputs the power consumption during this time period [9]. The output can then be further analyzed.

[1] http://www.statisticbrain.com/google-searches/
[2] http://www.giswatch.org/thematic-report/sustainability-climate-change/carbon-footprint-icts

More recently, a static analysis framework, based on Hoare logic with production rules, has been developed [13, 21]. The framework provides a developer with an upper bound on the energy-consumption of their software. This static analysis framework resulted in the implementation of the ECALOGIC[3] tool that, given a model of the hardware and a software implementation, can statically determine power consumption bounds [23]. Essentially, providing application developers with a straightforward method of determining the energy-consumption of their applications.

In order for ECALOGIC to run its analysis, it must be supplied with a software implementation and hardware models. The application is written in a '`while`'-type language called ECA that is annotated with loop bounds [13]. Similarly, each hardware component is modeled in a language called ECM. For example, for `DNS` server software, the component models represent the hardware that the `DNS` server software uses during its execution. These component models are populated with energy-consumption values that approximate real world energy-consumption. ECALOGIC gives developers an apparatus that can perform static analysis on software implementations that have been ported to the ECA language (along with its component models). However, it is a new tool and has not yet been used for analyzing nontrivial software applications.

This thesis sets out to research the limitations and possibilities of ECALOGIC by using the tool for analyzing the energy-consumption of `DNS` server software. The result of the analysis is an estimation on the energy-consumption of two `DNS` server implementations on a particular hardware server. This research focuses on using ECALOGIC to model and estimate the energy consumption of `DNS` servers once a query request has been received, the validation of these results once ECALOGIC has made its estimates and the evaluation of the measurements methodology and ECALOGIC. This thesis also looks into an independent research performed by `Remy Bien` and commissioned by `SURFnet`. The research presented in this thesis uses the same exact software as used by `Remy Bien` in his thesis.

1.2 Related work

Improvement of energy efficiency in electronics and computer systems is not a new area of research. Many research projects have been carried out in the realm of hardware and software design in order to improve energy efficiency. Especially seeing that energy consumption is a 'first-class system design constraint'. One example includes how the 'Intel Nehalem processors cannot simultaneously run all cores at the maximum frequency and voltage level without exceeding their power envelope' [24]. A less explored path has been that of how static analysis can help in making predictions about energy consumption. Such research investigates how compiler awareness of energy semantics can be developed for statically determining the energy consumption of applications.

In their paper 'Energy Types' [7], Michael Cohen et al. describe one such framework as a 'practical type system to reason about energy-aware software'. This type system is known as 'Energy Types'. The research developed a type system in an object-oriented language named `ET` that can be used for developing smart phone applications. The type system is one of the first to build energy management strategies into a programming language, and abstracts themes such as 'energy management as the reasoning of phases and modes'. The paper demonstrates that `ET` can lead to 'significant energy savings for an Android application'. A system called *Green* has also been developed on the programming language abstraction level and provides a 'simple and flexible framework' for supporting 'energy conscious programming using loop and function approximation' [2]. The difference between `Green` and `ET` is that the former does not make use of a type system to make its energy estimations.

Research on how to reduce energy consumption on the software abstraction level include inter alia, improving the operating systems scheduler for reducing CPU energy [29], research on how architectural designs can influence energy consumption [5], research on power aware page allocation and 'cooperative I/O queues hard disk accesses to maximize the standby time' [16] and how cache size and its power usage influence energy consumptions [24][26]. Another research path on reducing the energy consumption of software also includes research on compilers in [11][12][20][26]. In [20] the authors explore different compiler flags and their effect on minimizing the energy consumption for embedded

[3]The code is open source and can be found at: https://github.com/squell/ecalogic. A web version can be found at: http://ecalogic.cs.ru.nl/ecalogic-webapp/

platforms. They confirm that, generally, for many platforms the execution time and energy consumption are correlated[4]; a better set of flags, compared to the default ones, exists which produces more optical applications; and that it is possible to determine the effectiveness of each optimization.

Research on the performance of DNS server implementation are scarce, and research on their energy consumption is almost none existent. One paper that looked into the performance of DNS servers is 'A Performance view on DNSSEC migration' [17]. The authors of the paper, Daniel Migault et. al, research deployment of DNSSEC and performance aspects that provide 'experimental measurements for both DNS and DNSSEC architecture'. The paper looks into unitary tests (measurement without taking the server load into account), network latency, response times, update operation costs and the impact of cache hit rate. The research shows that for the unitary tests Unbound is found to be faster by 67% for DNS requests, 68% for DNSSEC without validation and faster by 46% for DNSSEC that also performs validation. The results for the maximum query load concludes that BIND is able to handle only 28% of the maximum query load that is handled by Unbound. Unbound also comes on top when comparing network latency and the impact of cache hit rate. The picture painted by [17] suggests that Unbound is more performant than BIND . It remains to be seen if this efficiency also translates to a lower consumption of electric energy. However, taking the results found in [20] into account, it is the expectation that Unbound would also consume less energy than BIND.

Lastly, in August of 2013 SURFnet put out an assignment[5] describing that the purpose of the task was to analyze different DNS resolvers in order to gain more insight on how these servers consume electric energy. The DNS resolvers to be analyzed were Microsoft DNS, BIND and Unbound. The assignment was ultimately carried out by Remy Bien, a former student of the Hogeschool van Amsterdam (HvA), between February 2014 up until the end of June 2014. The assignment replayed DNS queries that were captured from a live DNS server hosted by SURFnet , on a server in a test environment. Furthermore, Remy developed a measurement protocol that measures the energy consumption of the servers with an empty cache and a non-empty cache. The measurements were not only used to determine the most energy efficient server, but they were also used to calculate the biggest yearly monetary difference found between the three DNS servers. The results of the assignment is discussed more elaborately in Section 6.2 and Section 6.3 of this thesis.

1.3 Thesis outline

The structure of this thesis is as follows. Chapter 2 provides the necessary background needed for this thesis. The chapter discusses the Domain Name System, better known as DNS, introduces terminology used with DNS, its history, the problem it solves, the Internet and two particular DNS server software implementations and their differences in implementing the protocol. This chapter also introduces the energy analysis framework based on Hoare logic and ECALOGIC, an implementation of the framework in the form of a compiler. An example of a simple ECALOGIC program and its estimation results is given to provide the reader with a small introduction to ECALOGIC and how it works. Chapter 3 introduces the ECALOGIC models used throughout this thesis. However, these models do not yet contain the energy values needed for our evaluations. Chapter 4 describes the measurement methodology used to acquire the energy value needed to populate our ECALOGIC component models. The chapter also discusses another strategy taken that did not produce any meaningful results. Chapter 5 analyzes the measurements acquired from the server and populates our models from Chapter 3 with these values. This is followed by Chapter 6 in which the results of ECALOGIC are compared the measurements in Chapter 5 and the those found by Remy. In Chapter 7 the experience of working with ECALOGIC and the measurement methodology introduced in Chapter 4 are evaluated. Chapter 8 concludes the thesis and presents suggestions for future improvements for ECALOGIC.

[4]This finding will later be used in this thesis in order to fill in energy consumption values that could not be measured during the measurement sessions.

[5]http://www.sos.cs.ru.nl/applications/master/SURFnet_DNS_energy.pdf

Chapter 2

Background

This chapter discusses the background of DNS and introduces the energy analysis framework and its tool ECALOGIC. Section 2.1 starts by briefly explaining the history and functionality of name servers. This is followed by a short introduction to DNS ands some of its terminology. The section concludes by explaining how the best-known computer network, the Internet, uses name servers and the DNS protocol to make it operational. Section 2.3 discusses the energy analysis framework and its accompanying tool. This framework has been developed and presented in the papers 'A Hoare Logic for Energy Consumption Analysis' by Rody Kersten et al. and its accompanying technical report 'Soundness Proof for a Hoare Logic for Energy Consumption Analysis' by Paolo et al. Section 2.3.1 introduces the framework and gives a high level overview on how it functions, details the language grammar and introduces the notion of component models. In Section 2.4 an overview is presented on how the ECALOGIC tool is structured and a small example of how it can be utilized by developers is presented.

2.1 Domain Name System

A computer network connects different electronic devices (be it personal computers, servers, routers, etc.) with each other and has communication of data as its purpose. Each device is given a unique address where it is accessible for communication. This address can be thought of as the telephone number of the device. Therefore, accessing a device over the network requires knowing the numeric address of this resource, such as 173.194.65.101. With billions of devices connected to the Internet, and millions more being connected every day, it becomes impossible to remember such addresses. A system dependent on a *name server* was developed for relieving users of the network of having to remember long numeric addresses. The Domain Name System (DNS) uses these name servers to translate user-friendly domain names, such as 'google.com', to Internet Protocol (IP) addresses.

2.1.1 History of DNS

When networks were originally developed, each device on the network had a unique numeric physical address. However, humans have trouble remembering long numeric addresses compared to meaningful and descriptive names. This lead to the development of a system in which hosts are assigned names, making it easy for people to use the network [15].

One particular host on the network is responsible for maintaining a list of hosts and their associated names and addresses [15]. This designated host is known as the *primary name server*. Such a server makes it possible that other devices connected to the network are required to only know two things: the address of the name server, and the name of the host it wishes to communicate with (i.e. google.com). Devices on the network queries the name server for the address of a particular device. The name server answers, if possible, with the address of the desired host. Figure 2.1 depicts how this simple interaction is carried out.

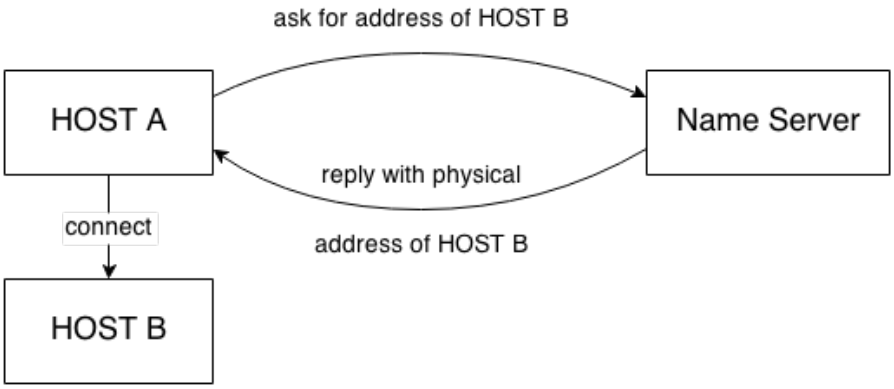

Figure 2.1: Model of name server in a network

2.1.2 Domain Name System

The primary goal of DNS is to provide a consistent way of providing access to resources [18][19]. It defines procedures for accessing data, referring to other name servers, caching retrieved data and the periodic refreshing of data. The system can be divided into three components that are important for its working, namely (i) the domain name space and resource records (ii) name servers and (iii) resolvers. Name servers and resolvers are often combined for efficiency reasons.

Domain Name Space

The domain name space is a hierarchical, distributed database that stores information used by Internet applications (also called 'clients') to retrieve information for them[1]. The database is represented as an inverted tree with the root node representing the '.' domain, followed by the *Top-Level Domains (TLDs)* and then by the *Secondary Level Domains (SLDs)*. The TLDs are split into two different categories: Generic Top-Level Domains (gTLDs) such as .com, .net, .org; and Country Code Top-Level Domains (ccTLDs) such as .us, .ca, .uk. Figure 2.2 gives a diagrammatic illustration of the tree. Each level of the hierarchy delegates responsibility of its domains to a lower level.

Resource Records

Resource records are answers to queries. They are data that is associated with a particular domain name. Every node in the three has a set of resource information (an empty set is allowed). This set of resource information, that is associated with a particular name, is composed of various resource records (RRs). It must be noted that the field RDATA is type, and sometimes class, dependent. For example, the RDATA of a RR with type 'A' only contains the physical 32-bit Internet address, i.e. `IPv4` IP addresses.

[1] ftp://ftp.isc.org/isc/bind/cur/9.8/doc/arm/Bv9ARM.ch01.html

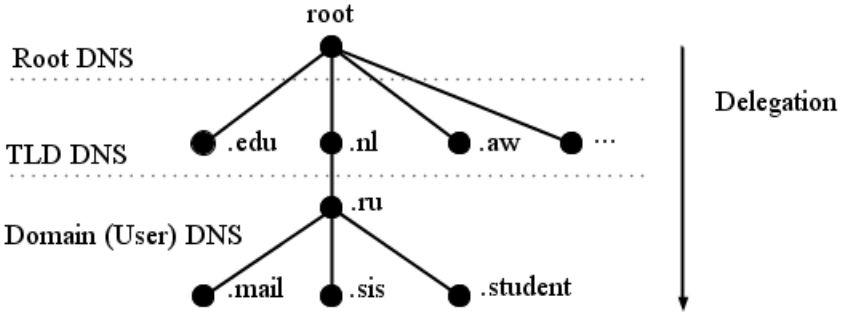

Figure 2.2: DNS hierarchy structure

Queries

Queries are messages sent to name servers by resolvers in order to retrieve information for a particular user. A standard query consists of a target domain name, the query type and the query class. The query asks for resource records that match. Name servers use information found in the fields to look for resource records that match the description.

Resolvers

Resolvers are responsible for querying domain name servers for data on behalf of user processes, such as Internet browsers, in order to provide them with the information they need (such as IP addresses). It is for this reason that RFC 1123 specifies that every host connected to a network must implement a resolver in order to work within the domain name system. Resolvers can be categorized in two types: full-service resolvers and stub-resolvers. A full-service resolver implements a complete resolver service (i.e. capable of dealing with following referrals, failures of name servers, etc.) and must contain a local cache. Stub resolvers are dependent on the capabilities of a recursive name server and only generate queries for user processes.

Name Servers

The main activity of a name server is to provide answers to standard queries requested by resolvers. Queries and their responses are both transmitted according to a standard message format. This message format is described in [19]. A response may consist of various resource records. How exactly a server responds to a query is dependent on how the server is configured. A name server can either be operating in a *recursive* or a *non-recursive* mode. A server operating in the non-recursive mode can only answer queries using local information. Therefore, the response will either be an error response, a proper answer if it is found locally or a reference to another server that might have the answer. Name servers operating in the recursive mode always respond with either an error message or an answer to the query. The server essentially works as a resolver in this mode and will always follow up on referrals to other name servers in order to provide an answer. RFC 1034 details how the algorithm for a name server is allowed to behave after receiving a query. The algorithm assumes that resource records are organized in different tree structures, one for the local cache and one for each zone.

2.1.3 The Internet

The Internet is the best-known computer network in the world. It is a global collection of networks connected to each other for the purpose of data communication. The Internet uses a specific implementation of the name server concept that has been optimized for usual conditions of the Internet [1]. It uses the DNS protocol for communication between networks that are connected to each other.

In order for Internet applications to retrieve the resources they need (such as images, text, video, etc.), they must know the IP address where theses resources are located. Imagine an Internet browser user that wants to visit 'www.google.com'. The browser uses the PCs stub resolver to find the IP address of the website[2]. The stub resolver creates a query and sends this query 'to its locally configured DNS resolver' [1]. In this example, the query target is 'google.com', its type is 'A' and its class is IN (the `INTERNET`).

The DNS resolver is expected to provide a response to the query. The server first consults its cache to find out if any resource records match the query when it is received. If a match is found, the server generates an answer for the query and responds with what was found. If a match is not found in the cache, the server will recursively search for an answer by first asking one of the thirteen root servers[3]. The root server responds with a list of referrals to authoritative gTLD name servers for the '.com' domain. Our name server selects a server from the received list and asks the same question to the selected gTLD name server. The gTLD name server in turn responds with a list of referrals to authoritative name servers for the 'google.com' SLD.

[2]Modern browsers usually implement their own DNS cache for a faster user experience.

[3]The Internet has a total of 13 root servers at the top of the DNS hierarchy located at different location around the world

The name server selects a server from the list it received and asks the same question to this SLD name server. The name server responds with the type 'A' resource record for 'google.com', which in this case is '173.194.65.101'. The name server sends this response to the stub resolver and the stub resolver informs the browser that the IP address for 'google.com' is '173.194.65.101'. The browser then sends a request to '173.194.65.101' for the web page.
The story depicted above is for when the query and the name server want a recursive solution to the answer. When the query is to be answered iteratively, a similar approach is followed. One difference however is that the DNS resolver will return the list of root servers if the answer is not found in its cache. However, in this case the PCs resolver is only a stub resolver. In such a case it is best that the configured DNS resolver supports recursive querying.

2.2 DNS Server Software

There exists different software implementations for the `DNS` protocol, such as `Microsoft DNS`, `BIND`, `Unbound`, `PowerDNS`, `NSD`, etc. This thesis focuses on the `BIND` and `Unbound DNS` server implementations.

2.2.1 BIND

The BIND DNS server software was first developed at the University of California, Berkeley, and is now maintained by the Internet Systems Consortium[4]. It is the most widely used DNS software on the Internet and is the *de facto* standard on Unix-like operating systems.

2.2.2 Unbound

Unbound is a validating, recursive, and caching DNS server implementation developed by NLnet Labs[5]. A prototype of the server was developed in 2004 in Java and in 2006 an implementation written in the portable 'C' language, based on ideas from the Java prototype, was developed. Contrary to BIND, Unbound is not a full fledged authoritative DNS server. The server does boasts higher performance and better security[6]. The latest version of this software will be used, namely version 1.4.21 released September 10, 2013.

2.3 Energy analysis framework and ECALOGIC

This section introduces the energy analysis framework and ECALOGIC presented in [13, 21] and [23] respectively. This section also provides a simple example in Section 2.4 on how ECALOGIC can be a benefit to programmers if they ever find themselves needing to choose between two different algorithms based on their energy efficiency.

2.3.1 Energy analysis framework

In [13] a method for modeling energy-aware systems with hardware and software components has been developed. This modeling technique is an energy-aware Hoare Logic that is sound with respect to an energy-aware semantics and can statically estimate the energy-consumption of software. This section will briefly describe the Hoare logic and further expand on how a hybrid system can be modeled. The latter is done by describing what the logic can do. The former is shown by giving a small overview on the language presented in [13] and showing how components can be modeled such as that the logic can make an approximation on the energy-consumption.

The frameworks semantics and its logic assumes that energy-aware component models are present within the system. These component models represent hardware or software components within the system that is to be measured. The analysis is performed on a hybrid system containing both the hardware and software models. The Hoare logic enables formal reasoning about energy-consumption by specifying judgments. The logic and its ability to reason about energy consumption allows the

[4] http://www.isc.org/downloads/bind
[5] http://unbound.net
[6] http://www.infoworld.com/t/applications/new-open-source-dns-server-released-599

framework to approximate an upper bound on the energy-consumption of the system, and all the components that are present during its analysis. Therefore, the framework can give an energy consumption bound for every component available.

2.3.2 Language

Software plays a vital role in controlling the hardware of a computer system. The energy analysis presented in [13] is performed on a 'while'-language with annotated ranking functions for while loops that is called ECA [21]. ECA is a simple language and therefore only supports the integer type; does not support global variables; parameters are always passed by value; and does not support recursive functions. The analysis framework uses ECA in combination with hardware models[7] to estimate the energy consumption of a particular program. Therefore, in order for a developer to determine an application's energy consumption, it follows that the application must first be ported to ECA. The language also introduces explicit statements for operations on the hardware components.

2.3.3 Modeling

Kersten et al. introduced a method for modeling the hardware components needed for its analysis. These models are known as component models. The hardware is modeled in such a way that the relevant information for energy-consumption analysis is provided. The model for the component effectively captures the behavior in terms of its effects on the program state.

The component model is an abstract model that can be instantiated for any component [21]. Two important aspects of the model are that everything is a component and the models cannot directly affect each other[8]. Each component inside the model consists of a component state $C_i :: s$. The component state is a collection of variables. A variable v in the state of a component C_i can be accessed with the expression $C_i :: s.v$. As it has been noted previously, variables are all of the integer type, also in the component state.

The analysis framework assumes that at least one component is always part of the environment, namely the `Implicit` component $C_{implicit}$. This `Implicit` component model is used to 'capture the effect of executing a statement' inside the language, such as arithmetical operations, assignments and control structures [13]. Therefore, this component model must have the resource consumption constants $C_{cpu} :: \mathfrak{E}_\alpha$, for the energy usage of α , and $C_{cpu} :: \mathfrak{T}_\alpha$, for the time it takes to perform α, where $\alpha \in \{e, a, w, ite\}$. The set α corresponds to the energy consumption for the evaluation of arithmetic expressions and integer comparison (e), assignments (a), while loops (w) and conditionals (ite).

2.4 ECALOGIC

ECALOGIC is an implementation of the energy consumption framework discussed in Section 2.3. Given a model of the hardware, as component models, and a software implementation in the input language, it is possible for the tool to statically determine power consumption bounds. Thus, providing application developers with a straightforward method for determining the energy-consumption of their applications. Figure 2.3 graphically represents how ECALOGIC is able to determine energy consumption bounds.

In order for ECA to estimate the energy bounds for an application, the developer must first develop the desired functionality of the application in ECA. Each hardware component is modeled in a similar language called ECM. Models of the hardware and a software implementation must be supplied to ECALOGIC. ECALOGIC then analyzes the software implementation with regards to the supplied component models and outputs its time and energy consumption bounds.

A part of the grammar for ECA is presented in Figure 2.4. The grammar implemented by ECA is different than the one presented in [13, 21]. In comparison with the grammar presented in [21], the grammar shown in Figure 2.4 does not have a return statement for functions.

[7] See Chapter 2.3.3.
[8] Later in this thesis we see how this can be detrimental when modeling applications.

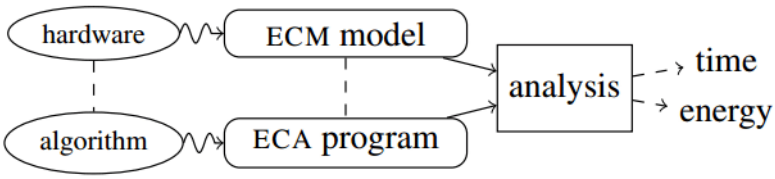

Figure 2.3: Schematic representation of ECALOGIC, taken from [23]

```
component Arr
    component function get(a, i)    uses 1 energy 1 time
    component function set(a, i, v) uses 1 energy 1 time
end component
```

Listing 2.1: ECM component implementation for an array type.

As an example of how ECALOGIC works and how it can be utilized by developers, the following use case is presented. Imagine a developer that must sort an array and must choose between two algorithms: `insertion sort` and `selection sort`. Both algorithms have a worst case of $\mathcal{O}(n^2)$[8], and both algorithms belong to the family of in-place comparison sorting algorithms [4][9]. The developer, however, is environment friendly and wants to take the energy consumption of his implementations into account and wants to pick the algorithm that consumes the least amount of energy.

In order to determine which algorithm consumes the least amount of energy, the developer decides to implement both algorithms in ECALOGIC. The developer uses the pseudo-code shown in Listing 2.2 as the reference for `insertion sort`, and its ECA implementation is seen in Listing 2.3. The `C` code shown in Listing 2.4 is used as the reference for the `selection sort` algorithm. Listing 2.5 shows the same algorithm implemented in ECA. ECALOGIC does not have a notion of arrays and, therefore, each developer developing in ECA must implement their own array component that simulates how arrays

[9] An in-place sorting algorithm sorts its elements with at most a constant size of additional storage for holding temporary values [8]

⟨*program*⟩ ::= {⟨*comp-imp*⟩ ⟨*sep*⟩} {⟨*fun-def*⟩ ⟨*sep*⟩}
⟨*comp-imp*⟩ ::= ‘`import`’ ‘`component`’ `id` {‘`.`’ `id`} [‘`as`’ `id`]
⟨*fun-def*⟩ ::= ‘`function`’ `id` [‘`(`’ [`id` {‘`,`’ `id`}] ‘`)`’] ⟨*fun-body*⟩
⟨*fun-body*⟩ ::= ‘`:=`’ ⟨*expr*⟩
| ⟨*stat-list*⟩ ‘`end`’ ‘`function`’
| ⟨*empty*⟩
⟨*stat-list*⟩ ::= {⟨*statement*⟩ ⟨*sep*⟩}
⟨*statement*⟩ ::= ‘`skip`’
| `id` ‘`:=`’ ⟨*expr*⟩
| ⟨*fun-call*⟩
| ‘`if`’ ⟨*expr*⟩ ‘`then`’ ⟨*stat-list*⟩ ‘`else`’ ⟨*stat-list*⟩ ‘`end`’ ‘`if`’
| ‘`while`’ ⟨*expr*⟩ ‘`bound`’ ⟨*expr*⟩ ‘`do`’ ⟨*stat-list*⟩ ‘`end`’ ‘`while`’
| ‘`{`’ ⟨*annot-elem*⟩ {‘`,`’ ⟨*annot-elem*⟩} ‘`}`’ [⟨*statement*⟩]
⟨*fun-call*⟩ ::= [`id` ‘`::`’] `id` ‘`(`’ [⟨*expr*⟩ {‘`,`’ ⟨*expr*⟩}] ‘`)`’
⟨*annot-elem*⟩ ::= `id` ‘`<-`’ ⟨*expr*⟩
⟨*expr*⟩ ::= ⟨*expr*⟩ ⟨*bin-op*⟩ ⟨*expr*⟩
| `id`
| ⟨*fun-call*⟩
| ‘`(`’ ⟨*expr*⟩ ‘`)`’
⟨*bin-op*⟩ ::= ‘`or`’|‘`and`’|‘`=`’|‘`<>`’|‘`>`’|‘`<`’|‘`>=`’|‘`<=`’|‘`+`’|‘`-`’|‘`*`’|‘`/`’|‘`^`’
⟨*sep*⟩ ::= ‘`;`’ | `end-of-line`

Figure 2.4: Grammar of the input language ECA, taken from [23]

N	Time	Energy	
		Implicit	*Array*
100	818805	808505	10300
200	3257605	3217005	40600
300	7316405	7225505	90900
400	12995205	12834005	161200
Complexity	$5 + 88N + 81N^2$	$5 + 85N + 80N^2$	$3N + N^2$

Table 2.1: ECALOGIC energy consumption estimates for `insertion sort`.

```
INSERTION_SORT(A, N)
    for j = 2 to N
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key
            A[i + 1] = A[i]
            i = i -1
        A[i + 1] = key
```

Listing 2.2: Insertion sort pseudo-code

work (see Listing 2.1 for the implementation used in this thesis). As one can see, ECA allows for a fairly one-to-one translation of these algorithms.

Let `A` be an array of length `N` and is always sorted in its reverse[10] order. This means that the algorithms always perform according to their worst case time. The ECALOGIC implementation of the algorithms are both run with four different array sizes, namely: 100, 200, 300 and 400. Table 2.1 and Table 2.2 show the energy consumption ECALOGIC estimates for the `insertion-` and the `selection sort` algorithm. The table also shows the complexity of the implemented algorithms. This complexity is obtained by running ECALOGIC with the variable N instead of a number. The results in the table show that the complexity class that algorithm belongs too is correctly given by ECALOGIC. The difference between two algorithms are the constants. Running the algorithms with different sizes arrays also show that time increases according to this complexity class.

Our developer can now objectively compare energy consumption analysis of his two algorithms. From the results table our developer decides to use the `insertion sort`. These examples show how ECALOGIC can be used to easily establish the difference in energy consumption of two distinct algorithms.

[10] `N` is the first element of the array and `0` is the last element

N	Time	Energy	
		Implicit	*Array*
100	930820	910520	20300
200	3701620	3621020	80600
300	8312420	8131520	180900
400	14763220	14442020	321200
Complexity	$20 + 108N + 92N^2$	$20 + 105N + 90N^2$	$3N + 2N^2$

Table 2.2: ECALOGIC energy consumption estimates for `selection sort`.

```
import component Arr
function insertion_sort(A, N)
    i := 1
    while i < N bound N do
        value := Arr::get(A, i)
        j := i - 1
        jvalue := Arr::get(A, j)
        while j >= 0 and jvalue > value bound N do
            Arr::set(A, j + 1, j)
            j := j - 1
        end while;
        Arr::set(A, j + 1, value)
        i := i + 1
    end while;
end function

function main(A, N)
    insertion_sort(A, N)
end function
```

Listing 2.3: The Insertion Sort Algorithm implemented in 2.4

```
void selection_sort(int arr[], int n) {
    int i, j, minIndex, tmp;
    for (i = 0; i < n - 1; i++) {
        minIndex = i;
        for (j = i + 1; j < n; j++)
            if (arr[j] < arr[minIndex])
                minIndex = j;
        if (minIndex != i) {
            tmp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = tmp;
        }
    }
}
```

Listing 2.4: Insertion sort pseudo-code

```
import component Arr
function selection_sort(A, N)
    i := 0
    j := 0
    tmp := 0
    minIndex := 0
    while i < (N - 1) bound N do
        minIndex := i
        j := i + 1
        while j < N bound N do
            if Arr::get(A, j) < Arr::get(A, minIndex) then
                minIndex := j
            else end if;
            j := j + 1
        end while;
        if minIndex <> i then
            tmp := Arr::get(A, i)
            Arr::set(A, i, minIndex)
            Arr::set(A, minIndex, tmp)
        else end if;
    end while;
end function

function main(A, N)
    selection_sort(A, N)
end function
```

Listing 2.5: Selection sort implementation in ECA

Chapter 3

Energy-Aware Modeling

This chapter discusses the ECALOGIC component models used in the rest of the thesis. Section 3.1 researches which hardware components are needed in general and which are necessary in order to model the `DNS` servers.

3.1 Component Models

In order for the Hoare logic to statically estimate the energy consumption, it must be given component models for which energy consumption information have been defined. Two different methods can be used to determine the required hardware components. The first method requires brainstorming on which server components may be used. The second method to determine the hardware components requires the utilization of the measurement equipment of `SEFlab` during the execution of a query request. The results of the former method have shown that three components are needed, namely a `CPU`, `Memory` and a `Network Interface Card (NIC)` component. One of the early preliminary measurement tests showed that, other than the `CPU` and `Memory`, no other measurable hardware showed an increase in their consumption.

The `NIC` is not modeled in this use case analysis, because component has not yet been equipped by `SEFlab` to have its energy consumption measured[1]. Future research with ECALOGIC on `DNS` servers should model this component to make the ECALOGIC analysis more accurate. Especially seeing how integral the `NIC` is for `DNS` servers. Thus, the only hardware components used during this thesis are the $C_{implicit}$ and C_{mem} component models.

The hardware used is more complicated than needs to be presented in the hardware models for ECALOGIC. This thesis looks at the hardware components as an abstract model and abstracts its energy consumption to its `idle` state, and the actual energy consumption of its component functions.

3.1.1 Implicit

Kersten et al. state in [13] that $C_{implicit}$ must always be present inside the energy model. The ECM component model declaration of our $C_{implicit}$ is seen in Listing 3.1 The `CPU` component model is kept simple and it will not have any state. The `CPU` also does not have any methods and, therefore, the model for $C_{implicit}$ will only consists of the set of constants needed by ECALOGIC.

It must be noted, however, that a realistic energy consumption component model of the `CPU` can never be modeled completely in the current form of ECALOGIC. The reason for this is that modern `CPU` do a lot of extra work, such as branch prediction, that the application developer does not have any control over. Furthermore, modern processors[2] often have multiple cores that, at the time of writing, cannot be modeled within ECALOGIC.

[1] This was planned as a future improvement to their measurement setup.

[2] For example, the processor used in this thesis has a total of four cores. All of which cannot be modeled using only $C_{implicit}$. Therefore, the measurements taken in this thesis only use one of the available cores.

```
component Implicit
    component function a    uses 5  energy 5 time
    component function e    uses 10 energy 10 time
    component function w    uses 25 energy 25 time
    component function ite  uses 25 energy 25 time
end component
```

Listing 3.1: Implicit component implemented in ECM

```
component Memory
    component function retrieve uses 5 energy 5 time
    component function store    uses 5 energy 5 time
end component
```

Listing 3.2: Meomory component implemented in ECM

3.1.2 Memory Model

The component model C_{mem} represents the memory model, i.e. RAM, found in the server. This model is shown in Listing 3.2. The model itself implements two functions: `store`, `retrieve`. Both functions take a variable name as parameter and both are used for storing and retrieving data from memory. The problem that a memory component model presents, is that its use must be explicitly stated in ECALOGIC. This means that each time a variable is assigned to, this must be followed by a call to $C_{mem} :: store$. Every reference to a variable must be preceded by $C_{mem} :: retrieve$.

3.1.3 Network Interface Card

The component model C_{nic} represents the servers `Network Interface Card`. This model is shown Listing 3.3 for future reference, but will not be utilized in this thesis. The model only implements one function $C_{nic} :: send$ that is responsible for communicating from the network.

3.2 ECA

This sections introduces the component models of both unbound (Section 3.2.1) and `BIND` (Section 3.2.2. As mentioned previously, this thesis only looks at the energy consumption of the servers when a query request has been received by the server. Thus, the approximation of the energy consumed of the server since it has been started is of no interest to this thesis[3].

3.2.1 Unbound

Two sets of component models have been developed for `Unbound` . The first set represents the cached component models and ECA source code. These are found in Appendix D.4, Appendix D.5 and Appendix D.6. The second sets represents the component models and ECA source code used for the uncached version of `Unbound`. These models are found in Appendix D.7, Appendix D.8 and Appendix D.9.

The architecture of Unbound consists of a 'core' implementation which supports modules that may or may not be initialized. Currently, the only modules available are the `validator` and `iterator` modules. The validator module is responsible for `DNS` query validation, according to RFC 4043, of

[3]Another project might only look at it from this perspective

```
component NIC
    component function send uses 50 energy 25 time
    component function retrieve uses 50 energy 25 time
end component
```

Listing 3.3: Network Interface Card component implemented in ECM

responses received from other name servers. The iterator module is responsible for performing the recursive iterative DNS query processing for answering a query. These modules are configured to be initialized by default inside the configuration file. The validator module is ignored for the rest of this thesis.

The modules are implemented as a state machines that can pass control of the thread to another module. An endless loop keeps track of the of which module has control of the thread and/or if the next module must be called. The iterator module is also implemented as an endless loop that switches from state depending on return values from its functions. This return values also decide if the its time to break out of this loop. The ECA logic code developed presents a simplified version of the state machine. Each state calls the functions it needs and decides what the next state must be. The state machine made the implementation in ECA straightforward.

3.2.2 BIND

Two sets of component models have been developed for Unbound . The first set represents the cached component models and ECA source code. These are found in Appendix E.4, Appendix E.5 and Appendix E.6. The second sets represents the component models and ECA source code used for the uncached version of Unbound. These models are found in Appendix E.7, Appendix E.8 and Appendix E.9.

The architecture of BIND is structured in the different functionalities it provides (such as the named server application, resolver library, etc.). Unlike Unbound, no module system is implemented in BIND. The implementation of BIND made its port to ECA somewhat difficult. BIND makes use of labels and goto statements, which, of course, are not supported by ECA. The code for BIND has thus been developed with help from the execution path retrieved from the logs. Conditionals have been used to simulate when certain code paths must be executed. A loop has been used to simulate recursion.

Chapter 4

Energy Consumption Measurement Methodology

ECALOGIC is dependent on input of energy consumption values for statements and constructs found inside ECA. The more these values approximate real world values, the more precise the energy consumption estimations of ECALOGIC will be. However, these estimations are specific to the underlying hardware on which the software is executed. Regrettably, most hardware components[1] do not come with their energy consumption values inside a manual[2]. This leaves researchers with no other choice but to acquire these consumption information in real world use cases.

In this chapter we develop a measurement methodology on how to obtain these energy consumption values by taking direct measurements from the hardware itself. This is done by first explaining the measurement environment, such as the hardware, the hardware measurement setup and the software setup. This is followed by a detailed explanation on how the measurements are performed and why this strategy has been selected. The chapter continues by showing how the energy consumption is calculated. Lastly, we end the chapter with a discussion on an alternate measurement methodology considered during this research.

4.1 Measurement Environment

The measurement environment was provided by the Software Energy Footprint Lab (SEFlab) at the Hogeschool van Amsterdam (HvA). SEFLab is equipped to take accurate measurements directly from hardware components found inside servers. This is achieved by equipping the server with power sensors on the hardware power lines. The setup can be seen as a black box that executes our software for a determined period of time on the server [9]. The energy consumption during this time period is the output of this black box. Analysis of the output is explained in Chapter 5.

Figure 4.1 displays a conceptional view of the SEFlab measurement laboratory [9]. Three components are important for the setup, namely the server where the software under test (SUT) is executed; the data acquisition (DAQ) hardware which collects measurements and provides them in a machine readable format; and the measurements computer where all measurements are recorded and analyzed. The server is an HP Proliant DL360 G7 integrated with an Intel Xeon Processor E5630 with 36GB (9x4GB) 1333MHz DDR3 of memory. The measurement computer is an iMac with the Windows operating system installed. The server consists of other hardware such as hard drive disks (HDD), NIC, a GPU, etc. Not all components are used for the analysis. The only components that are of any importance are the CPU and the RAM. As mentioned previously, the `NIC` is also taken into account when measuring however, `SEFlab` did not have the `NIC` equipped with measurement hardware.

One component not displayed in the diagram, but extremely important for our energy measurements, is the USB serial converter connected to the DAQ. The converter connects a USB port on the server to an independent channel on the DAQ. This extra connection makes it possible to a send electronic

[1] If not all components, such as the CPU, RAM, HDD, etc.
[2] Nor can one be found on the Internet

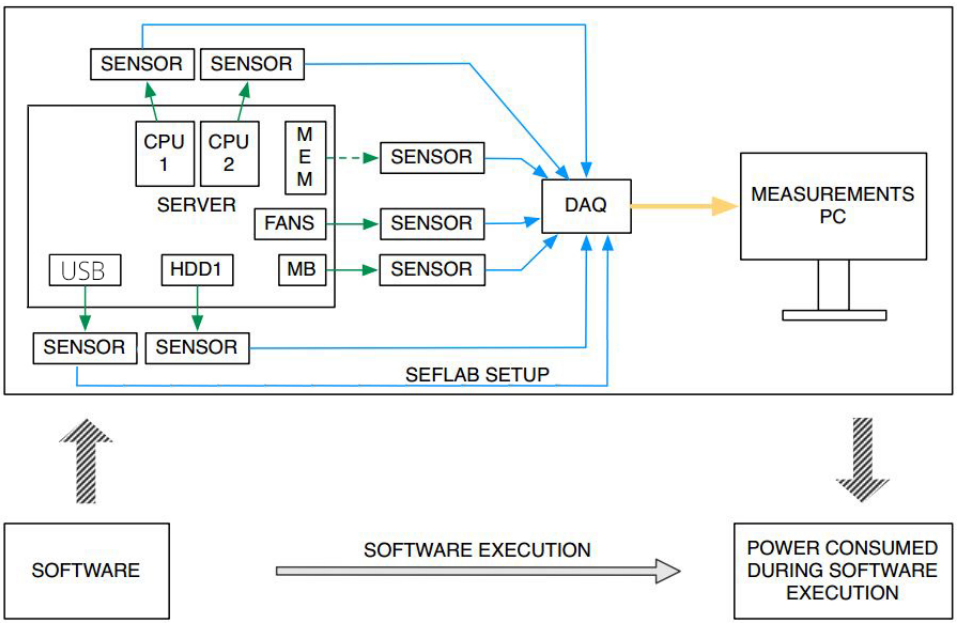

Figure 4.1: SEFLab measurement infrastructure, modified version of the one found in [9]

pulses to the DAQ. These pulses are sent before and right after our code under test is executed. The pulses are converted into timestamps that are later aligned with the power measurements once we have to analyze our measurements.

The operating system installed is CentOS 6.5, the `minimal install` version. The minimal install has the absolute minimum needed to have a functional system with no compromise regarding security. Our `DNS` servers, both `BIND` and `Unbound`, are installed on two different hard drives. Both `DNS` servers are configured to only operate on one thread and only listen to IPv4 `DNS` query requests made from `localhost`. Furthermore, the operating system was configured to use only 1 of the 4 available cores. How this is achieved can be seen in Appendix A. This configuration prevents the operating system from migrating the running code to a different core and affect our results.

4.2 Baseline energy consumption

Each component in the model uses energy, however, some component models are modeled after hardware that consumes energy when the operating system is `idle`. This energy consumption is called the `baseline` energy consumption. It is defined as the energy consumed by the component when it is not being utilized by the system.

This `baseline` energy consumption is acquired by letting the operating system run for ten minutes. Only processes that are started by the operating system when its booting are allowed to run during this period. No other applications started by the user should be utilized during this period. This will let us acquire the amount of energy the operating system consumes when it is idling in a realistic setting.

4.3 Implicit energy consumption

Chapter 2.3.3 explains that the energy framework assumes that there is always one component available, namely the `CPU` component model. This `CPU` component model is called `Implicit` inside ECA. A number of different applications have been developed in `C` in order to measure the energy consumption of the constants needed for the `Implicit` component model. A utility library, seen in Appendix B.2, has been developed in order to send a start and end pulse to the `DAQ`; time; and log the results of our expressions and language statements to a file on the file system. The utility takes measurements precise to the nanosecond. Furthermore, a script has been developed to build (Appendix B.3), run

and measure (Appendix B.4) each and every application under test. Appendix B.1 gives an overview on the structure of the directory and files that are important for the measurements.

4.3.1 Implicit template

A standard template, seen in Listing 4.1, was developed in order to acquire energy consumption values and is used by all of the C programs that have been developed. The template ensures that each measurement is executed using the same exact approach each time and to only differ in the initialization section and the body of the loop. Listing 4.1 is divided into three main sections. The first section imports the required headers, such as the timing utility functions (Appendix B.2). The second section is the `run_test` function. The last section is the `main` function and is the default entry point for any `C` program. The `POSIX` real time library is used for capturing the precise execution time of the statements that need to be measured. The total execution time of each run is logged to a file at the end of every execution run. This makes it possible the acquire the most accurate execution time, because its measured by the server itself.

The `main` function is responsible for initializing and closing the connection with the `DAQ`. A total of five measurements is performed by calling the function `run_test` with the number of instructions to execute as parameters. Each test doubles the number of instructions executed compared to the test that precedes it. This gives the possibility to verify that the execution time actually doubles on each run. This allows for the normalization of the energy consumption to a specific time unit such as `millisecond`. The measurements have for a small interval between each measurement in order to see a clear distinction between the tests in the graphs. The energy consumption graph for one such test is seen in Figure 4.2. The `run_test` function is were we perform our measurements and is responsible for initializing any variable needed by the body of the loop[3]. On example of such initialization is the addition of two variables `n` and `m`. Such an action requires that both variables are declared and initialized before the timer is started. Furthermore, this section is also responsible for determining the number of iterations the loop needs to perform. The number of iterations is determined by dividing

[3]The initialization section is different for each code under test.

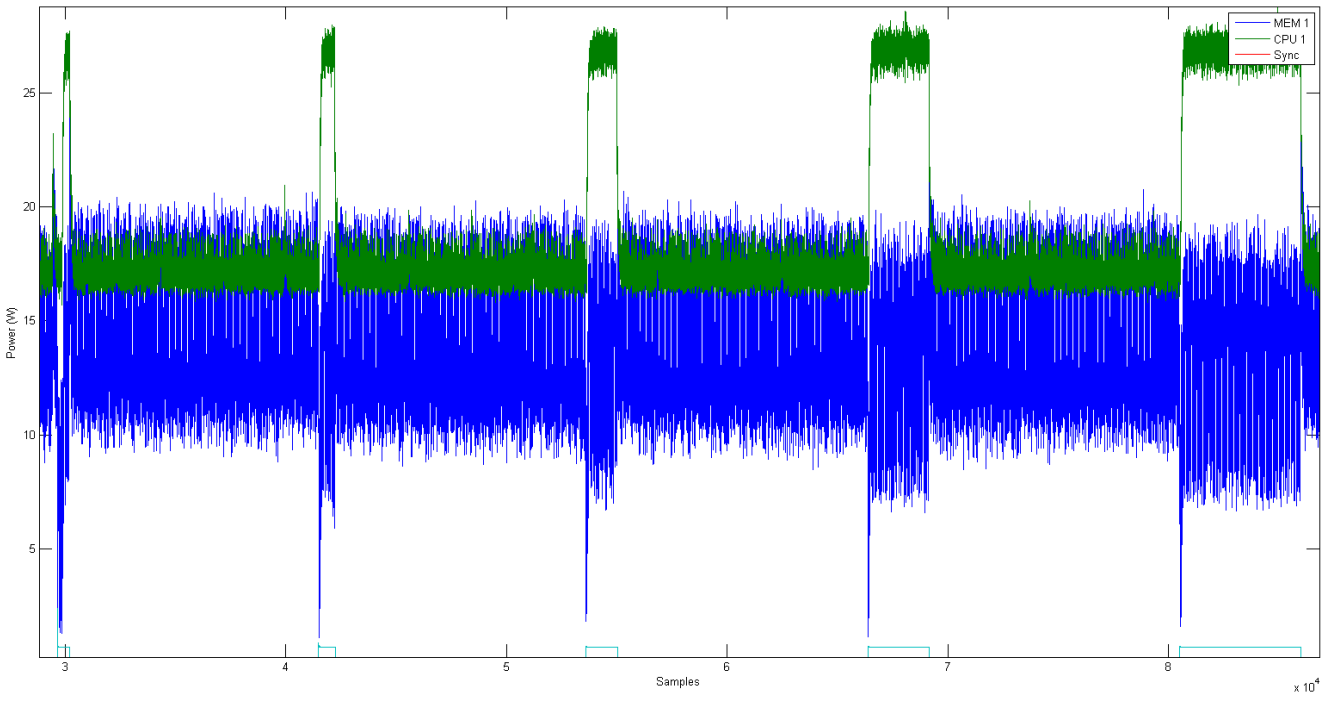

Figure 4.2: Energy consumption graph of five assignment test using the template from Listing 4.1

```
 // {# IMPORT HEADER FILES ... #}
void run_test(int loop) {
    // {# INITIALIZATION HERE #}
    set_rts(fd, &flags);
    start(&time);
        for(; i < iterations; ++i) {
            // {# 200 OCCURENCES OF THE STATEMENT #}
        }
    end(&time);
    unset_rts(fd, &flags);
    logtime("results/{# STATEMENT SPECIFIC FILE FOR LOGGING #}", &time);
}

int main(int argc, char** argv) {
    // {# INITIALIZE SERIAL PORT #}
    run_test(20000000);  usleep(sleep_time);
    run_test(40000000);  usleep(sleep_time);
    run_test(80000000);  usleep(sleep_time);
    run_test(160000000); usleep(sleep_time);
    run_test(320000000);
    // {# CLOSE SERIAL PORT #}
}
```

Listing 4.1: Measurement Template for Implicit

the `loop` variable by the number of statements executed inside the body (200) of the loop and is stored inside the variable `iterations`. The number 200 has been chosen in order to minimize the impact of our loop [27, 28]. Different number of statement executions have been tested and 200 has been the best fit in terms of overall speed. This also confirms the loop body size found in [28].

The `DAQ` channel is set to `high` before the timer is started and before the code under test is executed. This is followed by stopping the timer and setting the `DAQ` channel to `low` once the test application has finished running. The function `log_time` is then responsible for logging the results to a specific file on the server. Each statement measurement has its own log file were the results of the measurement can be found.

4.3.2 Compilation

All applications are written in the programming language `C` and compiled with the `gcc 4.6`[4] compiler. Unless otherwise noted, all code presented in the following sections are written inside the body of the loop inside the `run_test` function as presented in Chapter 4.3.1. All applications are compiled with the command line instruction seen in Listing 4.2. Appendix B.3 details the contents of the script.

```
gcc -O0 ./src/$1.c ./src/utils.c -std=gnu99 -o ./bin/$1 -I"./include/" \
    -lrt -Wall
```

Listing 4.2: Build settings for our measurement programs

Every application is compiled with no optimizations against the newest standard of C (gnu99) as implemented by the `GCC` compiler. `GNU99` provides interfaces specified by the POSIX standard, most notably the `Realtime Extension`. Furthermore, the standard allows for arbitrary definitions of variables inside functions. All variables must be declared at the beginning of the function if an older standard is used. The standard has no further implications on the code to be measured.

4.3.3 Component and ECA statement measurements

Main Memory The main memory of a computer is responsible for storing our program and the data they work on [25]. The `CPU` can perform two basic operations on the main memory, namely it retrieves instructions and data from main memory; and it can write the results of its calculations back to memory. This thesis takes the following view on these operations: storing a value in main memory is the equivalent of assigning a literal value to a variable inside programming languages; retrieving a

[4]This is the default `GCC` compiler version found on a clean installation of `CentOS 6.5`

```
// force the variable 'a' in register 'ebx'
register int a asm ("ebx");
...
// force the variable 'a' in main memory
volatile int a;
```

Listing 4.3: Build settings for our measurement programs

value from main memory is the equivalent of using said variable inside programming languages.

Six different scenarios have been identified on how the compiler, in this case `GCC`, compiles assignment and retrieval statements of data to and from a variable. Furthermore, its possible to either assign a variable to a variable or a literal expression to a variable. Let `a` and `b` be variables and that `b` is always assigned to variable `a`. The variables can be loaded from either the memory or a `CPU` register. The code snippet `'a = b'` (or `'a = 1'` when measuring the literal expression) is used to measure the energy consumption of the assignment statement. The following scenarios with the variables and literals are possible:

1. Both variable `a` and `b` are in the registers.
2. Variable `a` is in the register and variable `b` is in the main memory.
3. Variable `a` is in the main memory and variable `b` is in the register.
4. Both variable `a` and `b` are in main memory.
5. Variable `a` is in the register and is assigned a literal.
6. Variable `a` is in the main memory and is assigned a literal.

No measurements runs were performed for scenarios (2) and (3)[5]. THe compiler can be forced into setting the variable in either the register or main memory by using one of the code snippets found in Listing 4.3. Furthermore, scenarios one through six are executed twice with different values. The first time `b` has a value of `1` and the second time it has the value `INT_MAX`, which is `2147483647` with the current compile-time flags. The results of the last test may show if there is a correlation between the numbers of '0's and '1's inside the retrieved data.

Processor The processor is responsible for controlling the operations of the server [25]. It executes instructions that perform data processing functions on data retrieved from memory. The processor is represented by `Implicit` in ECA and requires consumption constants for expression evaluation, such as arithmetic and boolean expressions, and assignment statements.

In their paper, titled 'Comparative Analyses of Power Consumption in Arithmetic Algorithms Implementation', Alexandre Wagner C. Faria et al. mention that multiplication requires more energy than addition operations (independent of hardware). However, the latency of executing the different boolean operations and arithmetic operations are the same[6]. Therefore, only one expression is measured and this value is used as the constant for the component model.

The template is compiled with `n = a + b` inside the loop of the body in order to measure the energy consumption of the addition statement. This statement translates to assembly code that performs both an addition instruction and an assignment instruction. This assignment instruction is necessary because otherwise `GCC` optimizes away our code if it was just `a + b`[7]. The assignment statement forces the compiler not optimize away the addition instruction.

As with the assignment, different scenarios have been identified in which the variable used in a binary operation might find themselves. The scenarios are as follows:

1. Both variable `a` and `b` are in the registers.

[5] Of course, only one of these scenarios can be used for the component, however, it is still interesting to measure the effects of the different scenarios.

[6] See Table C-16, General Purpose Instructions, in the document titled 'Intel 64 and IA-32 Architectures Optimization Reference Manual'

[7] This is a default optimization and cannot be controlled by any the optimization flags

2. Variable `a` is in the register and variable `b` is in the main memory.
3. Variable `a` is in the main memory and variable `b` is in the register.
4. Both variable `a` and `b` are in main memory.

As with the `Main Memory`, measurements for scenarios (2) and (3) have not been performed. Its self-evident that point (1) and (4) already represent the lower and upper bound on the energy consumption, therefore, point (2) and (3) lies somewhere in between those points and the values are currently of no interest.

4.3.4 Language constructs

The following measurements concern measuring the overhead of certain concepts found in programming languages. These constructs are the `function call`, `while-loop`, and `conditional` (i.e. if-then-else) statements. All these constructs are a combination of different instructions found inside the `CPU` instruction set. Therefore, this thesis will look on how the energy consumption of the `CPU` can be measured when these constructs are executed.

Function call Functions are important for any programming language. They allow developers to break up their code into small manageable pieces of code. However useful they are, function calls do not come for free. Each function call performs the following steps: put variables on the stack; jump to the address of the function; load variables from the stack; perform instructions of the called function; clean the stack and then jump back to the function that originally called it.

Listing 4.4 shows the assembly instructions generated for an empty function call. Its possible to see that these instructions that empty functions still execute instructions[8]. These instructions are thus the energy consumption function overhead that each function call generates when its executed. Therefore, in order to measure the energy consumption of a function call, an empty function must be constructed and called inside the loop.

```
1  empty_function:
2  .LFB0:
3      .cfi_startproc
4      push    rbp
5      .cfi_def_cfa_offset 16
6      .cfi_offset 6, -16
7      mov rbp, rsp
8      .cfi_def_cfa_register 6
9      leave
10     .cfi_def_cfa 7, 8
11     ret
12     .cfi_endproc
13 .LFE0:
14     .size   empty_function, .-empty_function
15     .comm   fd,4,4
16     .section    .rodata
17     ...
18 .L5:
19     call    empty_function
20     ...
```

Listing 4.4: Empty function call `GCC` assembly

It is not necessary to put miscellaneous code inside the function, because `GCC` does not optimize away the empty function at this optimization level.

Loop Executing the same set of instructions inside programming languages is usually accomplished via recursion or loops. ECALOGIC, being a simple 'while'-language, opted to implement `while-loops` in its grammar and does not allow recursion. The energy consumption of a loop is measured by running the code snippet found in Listing 4.5. The `while-loop` cannot be measured the same way as the other code fragments. The boolean expression cannot simply be applied, because that will cause the loop to execute infinity. Therefore, the will count too 200. Allowing the outer loop to continue with it next

[8] If optimizations do not remove dead code

```
...
while(j < 200)
    ++j;
j = 0;
...
```

Listing 4.5: While-loop

iteration.

In order count with the `while-loop`, the integer variable must first be incremented. This operations consists of retrieving the value from memory, increment it and storing it back to memory. This in itself is not a problem because the time it takes to perform an addition and equality expression is known. These are subtracted from the total time in order to acquire the total time spent inside the `while-loop` for 200 iterations.

Conditional The conditional, also known as the `if-else` statement, has two paths that it can travel. The condition is either `true` and the 'if' branch is taken, or the condition is `false` and the 'else' branch is taken. The scenario and the code is almost identical, therefore, only the overhead of the energy consumed by the conditional when it evaluates to true is measured. The code that shall be used for the measurements is seen below in Listing 4.6.

```
...
if(ifi) { ++ifi; } else {}
...
```

Listing 4.6: While-loop

The compiler forces us to perform an action inside the body of the conditional, because otherwise it will be optimized away by `GCC` . This poses no problem for the analysis because its known how much time the addition instruction takes. These are subtracted from the total time in order to acquire the total time spent inside the `conditional` statement.

4.4 Measurement methodology DNS servers

In the end, the main objective of this thesis remains to see how well ECAlogic can estimate the energy consumption of DNS server software. This section lays out the measurement methodology used to acquire energy consumption values needed in the ECAlogic component models of the ECA `DNS` server code.

The analysis focuses on the time each server needs in order to answer an incoming query. Two scenarios on how the server can answer the query are considered, namely it finds the answer to the query inside its own cache; or the answer is not in its cache and the server must query root servers for the `IP` address. See Chapter 2.4 for a more detailed explanation on how the protocol works.

4.4.1 Measurement setup

Chapter 3 mentions that `BIND` and `Unbound DNS` server implementations will be examined. The first step was to install `BIND` and `Unbound` on two different hard drives. Secondly, the `/etc/resolv.conf` have been configured to only allow the use of `localhost` for resolving Internet domain names. Thirdly, both servers are configured to only listen for query request originating from `localhost` and request for `IPv4` addresses. Furthermore, both servers are configured to use recursion or iteration in order to answer any received queries. However, the servers are not configured to forward these queries. Lastly, both servers are configured to log its process each time a new query request is received. The complete configuration for both `BIND` and `Unbound` can be found in Appendix E.1 for `BIND` and Appendix D.1 for `Unbound`.

```
LOGGER("UnboundRBTSearch", UnboundRBTSearch);
e = slabhash_lookup(worker->env.msg_cache, h, &qinfo, 0);
END_TIMER(UnboundRBTSearch);
```

Listing 4.7: Build settings for our measurement programs

```
dig A google.nl @localhost
```

Listing 4.8: Command to send our queries to the DNS servers

Three different scripts have been developed for sending queries to the DNS server installed on the server. The scripts use `domain information groper` (dig) to send the queries. `Dig` is a network administration command-line tool for querying DNS name servers and can be installed by running the command `yum install bind-utils` on the command line.

4.4.2 Measurements

There are two scenarios in which the energy consumption for each server must be measured. The first scenarios measures the energy consumption of both servers when the answer is not found inside the cache. Followed by taking the energy consumption for each server when the answer is found inside their cache. In the first scenario a query request is send to the DNS server and instructs it to clear its cache afterwards. In the second scenario queries are send to the DNS server in order to warm up its cache. The time it takes to answer a query with a response found inside its cache is verified by comparing it to the time it took to answer the same query when a response is not found inside the cache. The scenario can also be verified by looking at the logs of each DNS server. The domain name that both DNS servers must resolve is `google.nl`. This domain name is chosen because its located in the Netherlands and, therefore, the response times will be smaller because of the locality of the server of Google. The query requests are send out by the command found in Listing 4.8. This command specifies that only responses with IPv4 IP addresses for the domain name `google.nl` are wanted, and also use the DNS server found at `localhost`.

The ECA component models of our servers needs energy consumption values before ECALOGIC can try to estimate its energy consumption. These models consists of different function calls that simulate the servers execution path. This means that measurements of these functions must be taken and the models will be populated with these values. The strategy used for measuring these functions is the same as the one used for the component models. Listing 4.7 shows one example of how such a function is measured. The listing measures the time and energy consumption needed for `Unbound` to search for a query in its cache. Both `LOGGER` and `END_TIMER` are helper macros defined to simplify taking measurements of function calls. The `LOGGER` macro uses the DNS server own logging mechanism to log to its file. The term `UnboundRTBSearch` is a custom macro defined for our purposes.

4.4.3 Compilation

In Chapter 4.4.2 an example of how to measure each function needed for the models is shown. The source of BIND and Unbound are modified with the timing utility code in order to determine the time spent in functions used to answer a query request. The timing functions have been slightly modified in order to make use of the servers logging mechanisms. As mentioned in the previous chpater, macros have been introduces to easily time these functions and log their execution time. These changes and the complete utility header can be seen in Appendix D.2 for `Unbound` and Appendix E.2 for `BIND`.

However, because of these change, both BIND and Unbound must be compiled and installed from source. Furthermore, the introduction of the timing library means that the build process for both DNS servers must be modified. This is done by modifying the needed `Makefile.in` files that control the build. Appendix D.3 and Appendix E.3 give a detailed explanation on how the build must be modified for Unbound and BIND in order to allow the use the timing functions.

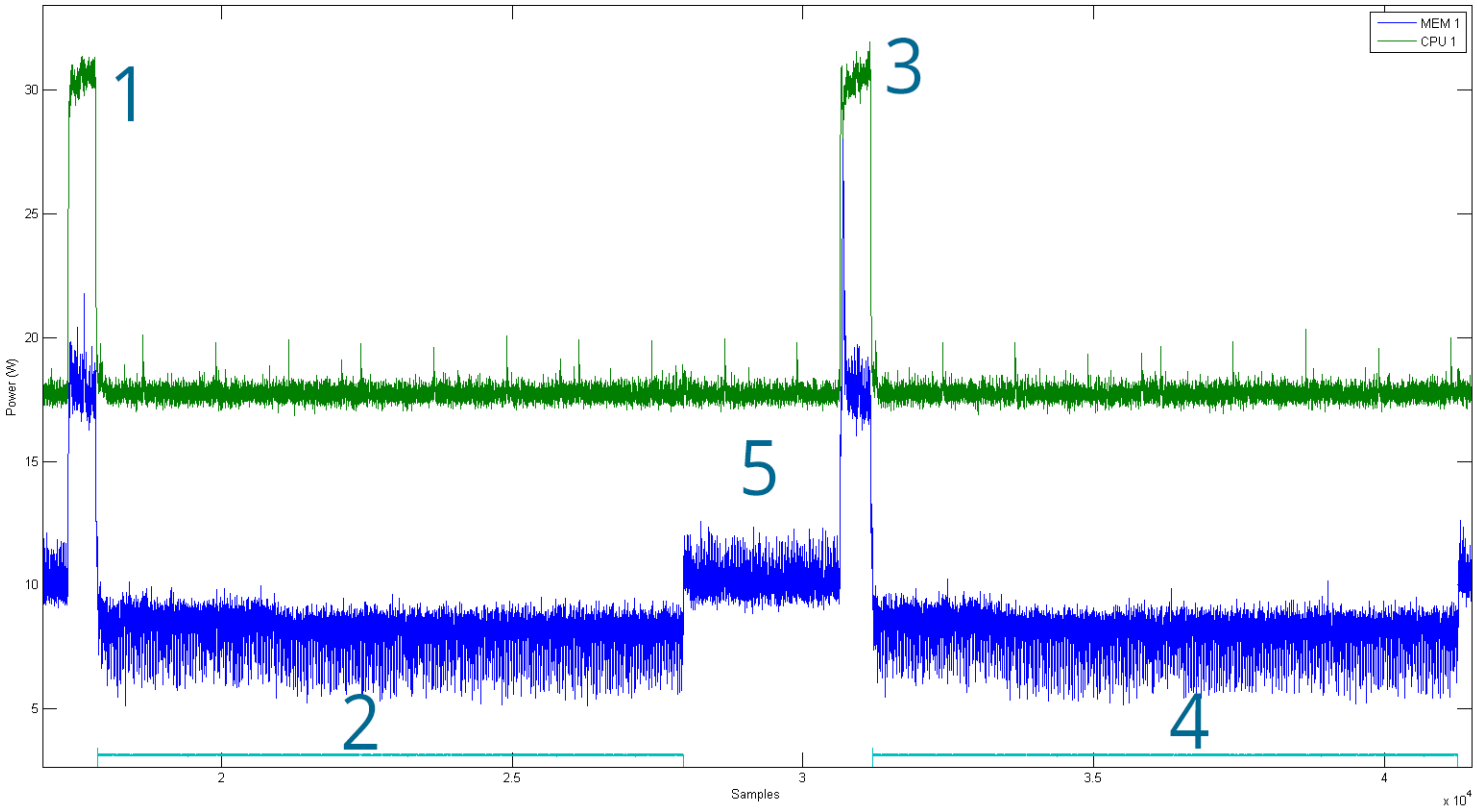

Figure 4.3: SEFlabs python script measurement

4.5 Unsuccessful approaches and final remarks

Initially another strategy than the one mentioned in the previous chapters was used for measuring the energy consumption of the assignment and binary statements. Originally the strategy involved executing one million, two million, four million and eight million assignment instructions one after another. However, this proved to be somewhat difficult. The compilation time of each program increased significantly each time the number of instructions doubled. Furthermore, the compiler crashed during the compilation of the four million and the eight million assignment instructions program. This lead to the reduction of the number of executed instructions, however, this did not work as expected. The execution time of the smaller programs was to small to capture any meaningful information. Another reason for this method not working, is that size of the total number of instruction that have to be executed is larger than any of the caches (L1, L2 or L3) and bigger than the cache lines (64Kb). Forcing the CPU to fetch the instructions it needs from memory.

Secondly, the method which SEFlab uses for their own measurements does not work at the level of time granularity this thesis needs for its measurements and analysis. SEFlab developed a python script that first sends a pulse for 50ms to the DAQ, starts the program under test, sends another pulse for 50ms and prints the running time on the console screen. Figure 4.3 shows the energy consumption graph for the CPU and Main Memory when this script was used for the measurements. There are five points of interest in this image. Number (2) and (4) represent the pulse that is sent at the start and end of the measurement. The figure shows that the energy consumption of the memory drops for the duration of the pulse period and increases again to its idle energy consumption at point (5). At point (1) and (3) its possible to see that the energy consumption of both the CPU and Main Memory increase right before sending the pulse to the DAQ. The python script uses a library for setting the pulse channel on high and low. Point (2) and (4) are set to high in the figure and the dip during point (5) is set to low. The program under test starts when the pulse is set to low. However, from the figure it is possible to see that there is a time period where both the CPU and Main Memory are idle before the energy consumption increases again at point (3). Be that as it may, a slight anomaly is seen right before sending the last pulse that is not seen in point (1). This anomaly can be observed more closely

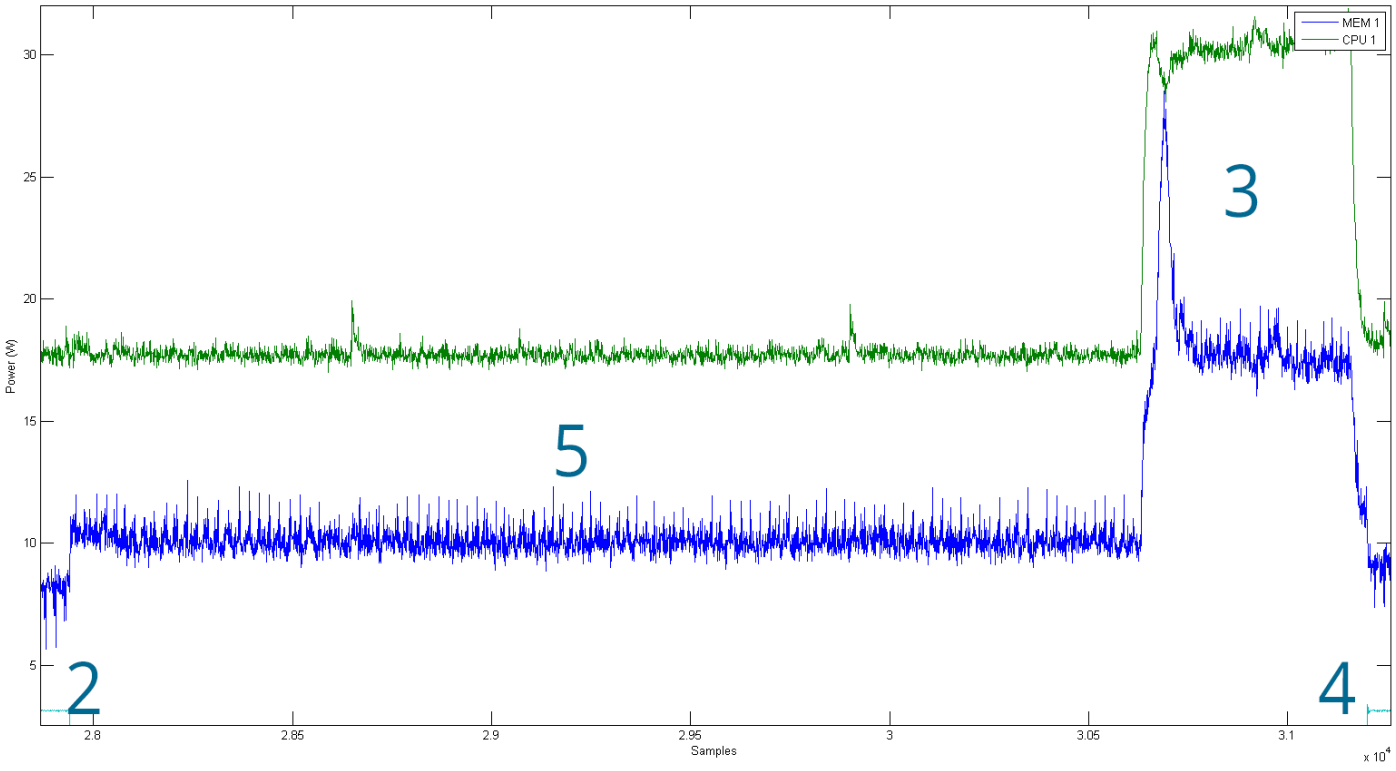

Figure 4.4: Measurement figure of assignment using `SEFlabs` python script

in Figure 4.4 and a closeup is found in 4.5. Furthermore, this anomaly seems to consistently take place during the preparation of sending the pulse. No satisfiable explanation for why this happens can be given. The code under test is found in Figure 4.5 by looking at where the `CPU` energy consumption drops and the energy consumption of the `Main Memory` increases. It has been verified that this section is the application under test, because an increase in execution time was seen for this section.

The measurement methodology described in the previous paragraph works great for `SEFlabs` measurement needs, because the time-scale they work with is larger than the one needed for this thesis. Furthermore, the extra overhead is insignificant for their experiments. However, this measurement methodology generates overhead that, in some cases, takes longer than the actual code that needs to be measured. This overhead and the anomaly observed, which cannot confidently be explained, means this method cannot be used reliably for the measurements of this thesis. This lead to the creation of the `C` template (see Listing 4.1) that talks directly to the `DAQ` and the methodology used in Chapter 4.3.

Lastly, acquiring an accurate energy consumption has its challenges brought on by the operating system. The operating system has its kernel space and user space processes that need to run in order to function properly. Therefore, the system scheduler must allocate time slices for each process to run. Each slice is then interrupted to give the next process in line its time to execute, this interruption is better known as the *context switch*. This switch undoubtedly affects our measurements because it takes time to switch the context (information of the current context must be saved and reloaded each time). This means that the energy consumption of this switch and other processes that occurred during this period are also measured. Thus, essentially providing a measurement that contains consumption of processes that are of no interest. Only energy consumed by the instructions and `DNS` servers are needed and not the consumption of any other process that the operating system is also running at the same time. `Linux`, and therefore `CentOS`, provides the user a way to designate one specific core to a specific application. Furthermore, the operating system can be configured in such a way that the scheduler does not utilize that one specific `core`. The steps needed to configure the operating system are documented in Appendix A. This setup would have allowed for the energy consumption of the processes to be *accurately* determined, because the energy measured could have only come from the processes that ran uninterruptedly on that single core. However, `SEFlab` can only measure individual

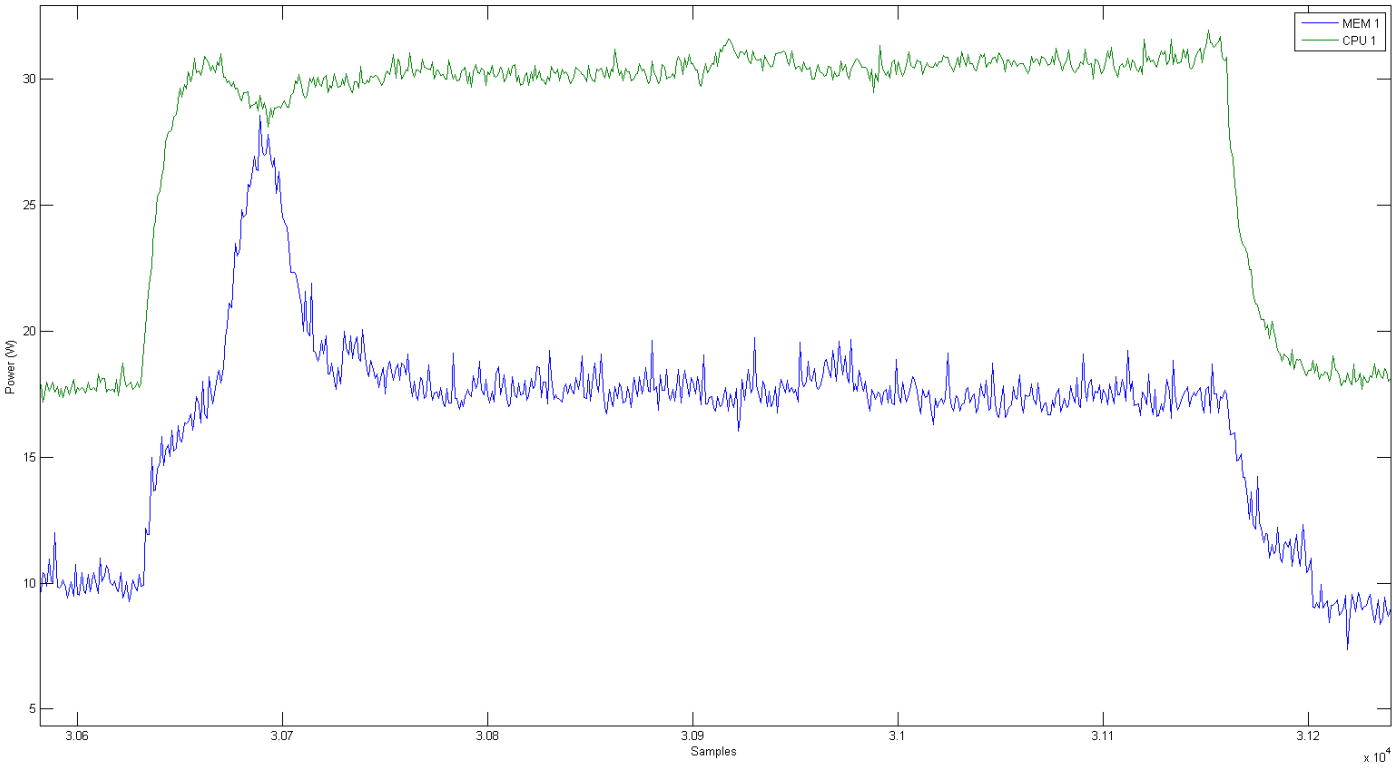

Figure 4.5: Measurement figure of assignment using `SEFlabs` python script

processors and does not, or cannot, differentiate between the individual cores found inside the servers processors. This means that this thesis can never be sure that our program under test ran without being interrupted[9].

[9]There exist other methods for ensuring that a particular program can run without being interrupted. However, this involves programming each program to be a kernel module and given a high priority. Nonetheless, most developers do not develop kernel modules and therefore the measurements presented here do represent the majority of code found in the wild.

Chapter 5

Energy Consumption Measurement Results

This chapter analyzes the energy consumption values acquired by using the measurement methodology discussed in Chapter 4. Section 5.1 explains how the measurement results for the analysis are analyzed. The section continues by detailing the energy consumption of the server `baseline` and the constructs needed for `Implicit` in ECA. Section 5.2 discusses the measurements for `BIND` and `Unbound` by showing the execution time for each function measured. In some cases this turned out to be the total execution time, in some cases it did not.

5.1 Measurement Results: Baseline and Implicit

Before analyzing the measurement results, the theoretical execution time limits of the `CPU` is determined. Knowing this information gives a view of the speed and total time the `CPU` needs to execute the number of instructions used later in this chapter. It has no further implications, but its an interesting piece of information nonetheless. According to the specifications[1], the server processor has a base frequency of 2.53 GHz. On the other hand, turning off the other three cores of the processor allows the remaining core to operate at a frequency of 2.8 GHz. The theoretical limits of the single running core of the processor at 2.53 GHz are displayed in Table 5.1[2]. For example, it should take the processor close to `7.91` milliseconds to execute 20 million instructions. As seen in Listing 4.1 (Section 4.3.1, page 30), the test is run fives times for each statement and the number of executed statements is doubled on each run of the test.

The `baseline` energy consumption of the server is determined by calculating the average energy consumption consumed by the server measured in each executed test (as discussed in Chapter 4). The `baseline` is measured by letting the server run when `idle` for ten minutes. The measurements are taken at a frequency of 10KHz. There are three scenarios for which measurements were taken for, namely the `baseline` consumption of the server; the `baseline` consumption of the server when `BIND` is running but is `idle`; and lastly, the `baseline` consumption of the server when `Unbound` is running but

[1] Intel® Xeon® Processor E5630: http://ark.intel.com/products/47924/Intel-Xeon-Processor-E5630-12M-Cache-2_-53-GHz-5_86-GTs-Intel-QPI

[2] The values shown do not take caches and modern `CPU` optimizations techniques into account.

Number of instructions (10e6)	Time (ms)
20	7.91
40	15.81
80	31.62
160	63.24
320	126.48

Table 5.1: Theoretical time needed by the `CPU` to run the number of instructions

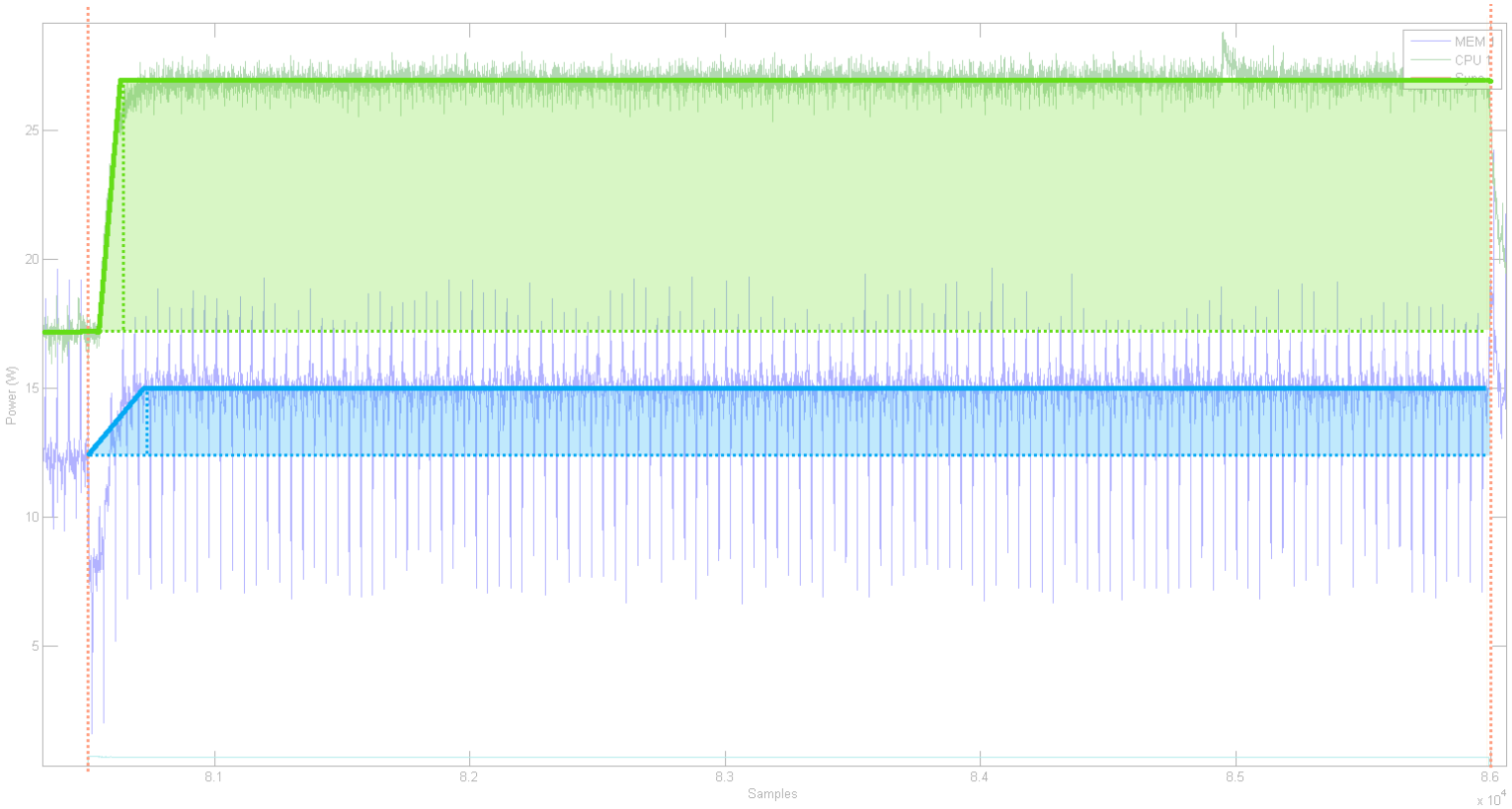

Figure 5.1: Graph for 320 **assignment** instructions showing the energy consumption for the `CPU` and `Main memory`.

is `idle`. The results of these measurements are discussed separately for each statement in Section 5.1.1.

Determining the energy consumption and total execution time for the `CPU` instructions such as assignments, conditionals, loops and binary operations is not as straight forward as that of determining the energy consumption of the `baseline`. Figure 5.1 shows an assignment statement that is executed 320 millions times inside the loop[3] (Listing 4.1). Figure 5.2 is a close up of Figure 4.2 that shows the start of that particular measurement. This figure will be used in the upcoming discussion on how the energy consumption is determined. The areas that are filled represent the energy consumption area that needs to be determined. The area on top is the total extra energy consumed by the `CPU` and the area below is that of the `Main Memory`.

As mentioned in Chapter 4, a `DAQ` is connected to the server that sends pulses towards the `DAQ` . This allows the time bounds in which the code under test was executed to be identified. The line at the bottom of Figure 4.2 and Figure 5.2 represents the pulse line. The `DAQ` channel for the pulse is set to high at the beginning of the test and is set back to low at the end of the test. This pulse line is also represented in Figure 5.1 by the leftmost and the rightmost dotted vertical lines. The figures show how the energy consumption of both the `CPU` and `Main Memory` increase in the time period in which the measurement took place. This pattern is observed for all the other measurements taken.

Working out the execution time of the tests can be accomplished by two different methods. First, there are the measurements taken from the hardware provided by `SEFlab` (the pulse line). For example, the time from Figure 5.1 is determined by identifying the sample when the pulse is set to high and the one for when it is set back to low. The number of samples taken during this time period is then multiplied by the frequency to calculate the total execution time. Section 4.3.1 discussed the second method for determining the performance of the server, i.e. the server takes its own performance measurements (as shown in Listing 4.1). The server starts its measurements after the start pulse has been send (line 4-5 of Listing 4.1) and ends it before the end pulse is send (line 9-10 of Listing 4.1). Thus, sending

[3]This is a zoomed in version shown of the last test run seen in Figure 4.2.

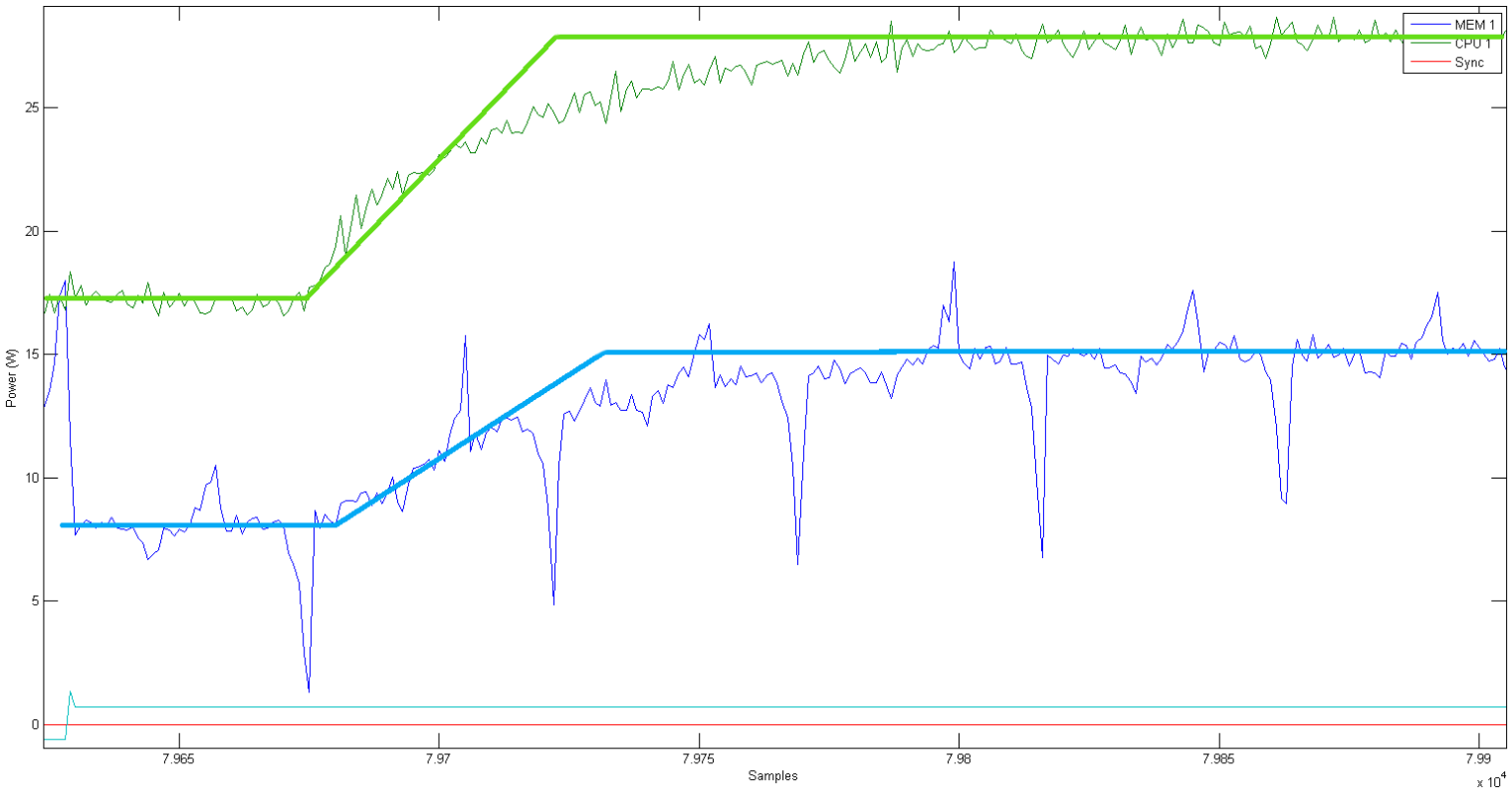

Figure 5.2: Close up of Figure 5.1 at the beginning of the measurement.

the start and end pulse to the DAQ, the first method of taken the execution time, is also taken into the energy consumption measurements taken by the SEFlab hardware. Hence, the execution time taken on the server itself is always the most accurate measurement. These values are therefore used by the component models found in Appendix D and E. The expectation is that the execution time measured by SEFlab is always higher than those taken by the server[4].

The next step is to determine the difference in energy consumption witnessed in Figure 5.1 and Figure 5.2. Of course, the consumed energy during the sample period is continuous but not constant. Not only that, as seen in Figure 5.2, there is a period were the energy must transition from the baseline consumption level to a higher state of energy consumption. The area below the lines is determined by first calculating the mean and standard deviation energy consumption of the higher state. This is followed determining the area below the triangle formed when going from the baseline state to the higher state.

First, the histogram for the sample period is determined. The histograms bins are decided by rounding down the sampled values to one decimal point[5]. The histogram provides an overview on the most frequently encountered measurement values. Therefore, the average energy consumption should be seen more frequently. The most frequent histogram value is validated when the mean of the higher state is determined. These values should either be the same or not too far from each other. The most frequent value is used in combination with a deviation, in this case 1, in order to discover the position where the energy consumption values first starts to fall within the limit. The first value that falls inside this limit is marked as the start of the higher state. The same idea is used for obtaining the end of the

[4]This is verified later in Section 5.1.2 when the actual measurements are analyzed.
[5]These values go up to the 16^{th} decimal point, however they are only reliable up to the 12^{th} decimal point.

$$A = \frac{1}{n}\sum_{i=1}^{n} a_i \qquad (5.1)$$

$$s_N = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2} \qquad (5.2)$$

	baseline	
	CPU	*Memory*
1st Run	17.421753310000	9.266251231667
2nd Run	17.428363491667	9.271000996167
3th Run	17.431117228333	9.271255357833
Average:	17.427448343333	9.271255357833

Table 5.2: Baseline measurement and average for the server and DNS servers

higher state by using the end pulse. The algorithm is designed to backtrack once the end of the pulse is reached. The algorithm backtracks until the first value falls within the limits. This sample marks the end of the area that needs to be determined. The higher state `mean` and the `standard deviation` is then calculated from the values found between these bounds. The value found for the `mean` is then multiplied by the number of samples to find the total energy consumed. The `mean` is determined with equation 5.1 and the `standard deviation` is computed with equation 5.2.

Second, the first point found in the previous calculation is consequently used the determine the difference between this point and when the pulse started. This variation is than multiplied by the difference between the higher state and the `baseline` and divided by two. Equation 5.3 is used in order to calculate the energy consumption of the triangle shown in Figure 5.1 by the dotted lines.

$$T = \frac{1}{2}bh \tag{5.3}$$

Initially it was the intention to add the previous two steps together in order to determine the energy consumed by the triangle and the rectangle. This value was to be used as the total energy consumed by the code under test. However, the triangle base turned out to be a constant in every measurement, which led to the assumption that this is the energy needed to go from the base state to a higher power state and is therefore not needed by our analysis.

5.1.1 Baseline

The results of the `baseline` energy consumption are summarized in Table 5.2, Table 5.3 and Table 5.4 respectively for the server baseline; the server running `Unbound`; and the server executing `BIND`. Comparing Table 5.2 with Table 5.3 and Table 5.4 shows that the average energy consumption of the `baseline` is below the energy consumption of both `Unbound` and `BIND`. This is an expected result, because the server is performing more work when running the `DNS` servers. Additionally, comparing the average consumption between `Unbound` and `BIND` reveals that the `CPU` consumes less energy with `Unbound` than `BIND` when the servers are in the `idle` state. On the other hand, the averages of the memory show that `Unbound` consumes more energy than `BIND`.

	baseline UNBOUND	
	CPU	Memory
1st Run	17.435579417637	9.276596497837
2nd Run	17.431702985000	9.274902912333
3th Run	17.442043279391	9.279161330982
Average:	17.436441894010	9.276886913717

Figure 5.3: Baseline measurements and averages for the `Unbound` DNS server

	baseline BIND	
	CPU	Memory
1st Run	17.454938501667	9.271220210667
2nd Run	17.459033565000	9.275302389500
3th Run	17.459982443333	9.272233761333
Average:	17.457984836667	9.272918787167

Figure 5.4: Baseline measurements and averages for the `BIND` DNS server

5.1.2 Implicit

This section discusses the analyses for the `Implicit` component model by showing the energy consumptions graphs and energy consumption values for each executed test. The tests for assignment and binary statements were run for multiple scenarios, as mentioned in Section 4.3. These differences will be shown in the corresponding energy consumption tables. Each implicit statement analysis starts on its own page due to space considerations.

This section also confirms that, as mentioned previously, the timing measurements taken by the `SEFlab` instruments should always be more than those taken by the server itself. The measurements show, were the data is available, that the `SEFlab` time is always 1 to 2 milliseconds longer.

Display analysis

The statistics and analysis of each measurement (i.e. the assignment, binary, etc) is shown in three different tables. The first table shows the measurements of each executed test retrieved from the server. The table also shows the difference multiplier needed to go from one run to the other. This difference is shown as values under the column $\xrightarrow{\pm\times}$. The symbol $\xrightarrow{\pm\times}$ signifies that the value in the column is the approximated value needed to get the value in the next column. This is done to validate that the execution time doubles with each run, i.e. it behaves as expected. Therefore, these values should be close to double the value of the previous.

The second table shows the values acquired from the `SEFlab` measurements equipment. The first and second column of the table shows the start and endpoints of the sync pulses. The third column is the difference between the first two columns. The fourth column represents the total time in milliseconds it took to execute between the pulses. The expectation is that the values in this column are always more than those found in the first table, because `SEFlab` is also measuring the time for sending the signals to start and end the pulse. The last column in the table show the difference between the current row and the previous row. This is comparable to the $\xrightarrow{\pm\times}$ symbol discussed above.

The third table shows the total energy consumption of the measurements taken for that particular statement. The table also shows the `mean` , `standard deviation` and the `variance` of the higher state. Figure 5.1 and Figure 5.2 show that the energy consumption is more volatile than that of the `CPU` and, therefore, a higher `standard deviation` is expected.

Assignments

As detailed in Chapter 4.3.3, tests for the assignment statement have been executed multiple times in order to see the effects of variables that are residing in the CPU cache or main memory. Table 5.3 shows that assigning literals to the register of the `CPU` is the fastest operation. The table also shows that assigning the value of one variable to another is slower than assigning literals. Even the assignment of a variable from one register to another more than doubles the execution time. The values of the row $a_{mem} = b_{mem}^{max}$ are used for the component model.

Binary operations

The energy consumption of each instruction executed by the CPU is dependent on the number of 1s presented in the opcode. Furthermore, the number of 1s in the operand of the instruction can also influence the energy consumption. Read 'An accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors' [14].

Function calls

Function calls are ideal for splitting a complex application up in manageable parts. However, as seen in Table 5.9, these calls do not come for free. The execution times of `function calls` do not need to subtract anything from its performance. As mentioned in Section 4.3.4, no 'binary' and 'assignment' statements had to be placed inside the function, because `GCC` does not remove an empty declaration at this optimization level.

Conditionals

The conditional statement, unlike a function call, did have to perform a 'binary' and 'assignment' operation. Therefore, the total execution time of the 'binary' statement has to be subtracted from values found in the row `IFT` of Table 5.12.

While-loops

As with the 'conditional' statement, the 'while-loop' also needed to perform a 'binary' and 'assignment operation. Therefore, the total execution time of the 'binary' statement will be subtracted from the values found in the row `WHILET` of Table 5.15.

	$20mil$	$\xrightarrow{\pm\times}$	$40mil$	$\xrightarrow{\pm\times}$	$80mil$	$\xrightarrow{\pm\times}$	$160mil$	$\xrightarrow{\pm\times}$	$320mil$
$a_{mem} = 1_{lit}$	7.588938	1.994	15.132962	1.997	30.219163	2.002	60.504637	1.998	120.890088
$a_{mem} = INT\_MAX_{lit}$	7.682238	1.967	15.13125	1.998	30.225473	1.999	60.406026	2.001	120.885101
$a_{mem} = b^{max}_{mem}$	7.701993	1.994	15.358931	2.004	30.771551	1.992	61.318672	2.000	122.664805
$a_{mem} = b^{1}_{mem}$	7.79897	1.969	15.359777	1.997	30.678779	1.997	61.274191	2.001	122.659511
$a_{reg} = 1_{lit}$	2.572832	1.98	5.09774	1.992	10.155476	2.005	20.364137	1.989	40.506798
$a_{reg} = INT\_MAX_{lit}$	2.574202	1.97	5.094709	1.992	10.146604	1.997	20.261365	1.998	40.488136
$a_{reg} = b^{1}_{reg}$	5.075665	1.99	10.106895	1.995	20.165865	1.998	40.284595	2.002	80.64489
$a_{reg} = b^{max}_{reg}$	5.07916	2.00	10.206729	1.976	20.165844	1.999	40.305267	2.000	80.624357
eca	7.701993	1.994	15.358931	2.004	30.771551	1.992	61.318672	2.000	122.664805

Table 5.3: Server execution time of **assignment** statements

	Energy (J/Instr)(10^{-5})	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	15.9933	3.8397	2.604
Memory	9.4105	3.8397	2.604

Table 5.4: Total energy consumption **assignment**

	Start	End	Δ	Time	$\xrightarrow{\pm\times}$
1^{st} run	25660	26227	567	12.6	
2^{nd} run	37507	38261	754	16.756	1.329806
3^{th} run	49602	51042	1440	32	1.909814
4^{th} run	62384	65174	2790	62	1.935700
5^{th} run	76515	82006	5491	122.022	1.968100

Table 5.5: `SEFlab` meta data information of **assignment** statements

	$20mil$	$\xrightarrow{\pm\times}$	$40mil$	$\xrightarrow{\pm\times}$	$80mil$	$\xrightarrow{\pm\times}$	$160mil$	$\xrightarrow{\pm\times}$	$320mil$
$a_{mem} + 1_{lit}$	45.186281	1.997	90.237125	1.999	180.341187	2.000	360.737157	2.000	721.391655
$a_{mem} + b_{mem}$	45.092583	2.001	90.241323	1.998	180.32607	2.003	361.23721	1.997	721.42770
$a_{reg} + b^{1}_{reg}$	7.553266	1.994	15.060914	1.999	30.102288	2.000	60.19981	1.997	120.172524
$a_{reg} + b^{max}_{lit}$	7.551816	1.994	15.055053	1.998	30.075312	2.002	60.203944	1.998	120.284149
eca	37.039059	2.022	74.882392	1.997	149.554505	2.005	299.918533	1.996	598.762894

Table 5.6: Server execution time of **binary** statements

	Energy (J/Instr)(10^{-5})	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	81.4350	18.7177	0.443
Memory	48.5577	18.7177	0.443

Table 5.7: Total energy consumption **binary**

	Start	End	Δ	Total Time	diff
1^{st} run	22775	25031	2256	50.133333	
2^{nd} run	36332	40462	4130	91.777778	1.830674
3^{th} run	51803	59994	8191	182.022222	1.983293
4^{th} run	71335	87671	16336	363.022222	1.994384
5^{th} run	99013	131551	32538	1723.066667	1.991797

Table 5.8: `SEFlab` meta data information of **binary** statements

	$20mil$	$\xrightarrow{\pm\times}$	$40mil$	$\xrightarrow{\pm\times}$	$80mil$	$\xrightarrow{\pm\times}$	$160mil$	$\xrightarrow{\pm\times}$	$320mil$
eca	32.743288	2.018	66.089219	2.029	134.101731	1.971	264.320147	2.001	529.111758

Table 5.9: Server execution time for **function** calls

	Energy (J/Instr)(10^{-5})	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	99.9164	16.5849	0.603
Memory	43.2519	16.5849	0.603

Table 5.10: Total energy consumption: **functions**

	Start	End	Δ	Total Time	diff
1^{st} run	22418	24120	1702	37.822222	
2^{nd} run	35424	38485	3061	68.022222	1.798472
3^{th} run	49826	55947	6121	136.022222	1.999673
4^{th} run	67288	79259	11971	266.022222	1.955726
5^{th} run	90600	114497	23897	531.044444	1.996241

Table 5.11: `SEFlab` meta data information of **function** calls

	$20mil$	$\xrightarrow{\pm\times}$	$40mil$	$\xrightarrow{\pm\times}$	$80mil$	$\xrightarrow{\pm\times}$	$160mil$	$\xrightarrow{\pm\times}$	$320mil$
IFF	16.529986	1.996	32.989696	2.003	66.068826	1.999	131.996	2.000	264.007
IFT	56.478069	1.999	112.891512	1.995	225.204552	2.000	450.423	2.000	900.625
eca	11.385486	1.989391	22.650189	1.981373	44.878482	1.987272	89.18579	2.00925	179.1973

Table 5.12: Server execution time: **conditionals**

	Energy (J/Instr)(10^{-5})	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	71.9033	17.2932	0.266
Memory	44.7414	17.2932	0.266

Table 5.13: Total energy consumption: **conditionals**

	Start	End	Δ	Total Time	diff
1^{st} run	28355	31124	2769	61.533333	
2^{nd} run	42407	47527	5120	113.777778	1.849043
3^{th} run	58868	69084	10216	227.022222	1.995313
4^{th} run	80425	100767	20342	452.044444	1.991190
5^{th} run	112108	not vailable	not vailable	not vailable	not vailable

Table 5.14: `SEFlab` meta data information of **conditional** statements

	$20mil$	$\xrightarrow{\pm\times}$	$40mil$	$\xrightarrow{\pm\times}$	$80mil$	$\xrightarrow{\pm\times}$	$160mil$	$\xrightarrow{\pm\times}$	$320mil$
$WHILET$	53.521318	2.000	107.049075	2.000	214.073906	1.999	428.014755	2.000	856.130674
$WHILEF$	7.703488	1.993	15.356762	1.998	30.67815	2.002	61.4053	1.998	122.666648
eca	8.428735	2.022	16.807752	1.997	33.74785	2.005	66.77755	1.996	134.702975

Table 5.15: Server execution time: **while**

	Energy (J/Instr)(10^{-5})	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	68.2897	16.2301	0.280
Memory	42.1377	16.2301	0.280

Table 5.16: Total energy consumption: **while**

	Start	End	Δ	Total Time	diff
1^{st} run	24176	26811	2635	58.555556	
2^{nd} run	38127	43032	4905	109	1.861480
3^{th} run	54373	64094	9721	216.022222	1.981855
4^{th} run	75436	94788	19352	430.044444	1.990742
5^{th} run	106129	not vailable	not vailable	not vailable	not vailable

Table 5.17: `SEFlab` meta data information of **while** statements

5.2 Measurement Results: Bind and Unbound

This section discusses the values measured at SEFlab for BIND and Unbound. The component models shown previously for both DNS server implementations are filled with the values presented in the tables found below. Statistical interesting values such as the mean, variance, and standard deviation are shown for both the cached and non-cached queries in the Appendices.

Regrettably, due to either hardware failures, configurations settings and/or inconsistencies, only the execution time found inside the logs of both BIND and Unbound can be used. This means that only values for the execution time shown in the tables below actually represent real world values. One assumption has to be taken in order to determine the consumed energy: both the CPU and Memory has two states, namely idle and the higher state. The energy consumption is determined by multiplying the mean value by the average energy consumption (the baseline is subtracted from this value) of all the statements measured in previous section.

The first three columns of the tables show the execution time for each measured range with the fourth column showing the mean of the first three columns. Each range has been given a unique name that encompasses the functionality its abstracting. The values of each column are summed up at the bottom of the table[6]. The last two columns represent the energy consumption of the CPU and Memory in Joules based on the assumption taken above.

5.2.1 Measurement Results: BIND

The measurement results for each measured range that is executed during the execution path when looking up cached and non-cached domain names are shown in Table 5.18 and Table 5.19. The total execution time for each run is totaled at the bottom of the table. Table 5.18 and Table 5.19 demonstrate that, for the largest part of the execution path, the same code ranges are encountered. The table also shows that the majority of these execution points take about the same time to execute. However, a few execution points show an increase (ParseRequest, RpzZones, CleanUp) and others, such as AddAuth, are not executed at all.

The timing results table also show that the server consumes more energy in almost every measurement point. Of course this is expected, because the server must carry out more work. However, because BIND executes the same path for a cached or non-cached request, in the end this means that each domain query request significantly increases the energy consumption (as can be seen in the result table).

Measurement anomalies

A measurement mistake has been identified during the analysis of the measurements. Table 5.18 shows that AccessLevel does not take any time since the value is zero. However, the value is zero because ParseRequest has been measured twice. The measurements of ParseRequest is not affected, mainly because the second measurement overwrites the first. Nonetheless, the timing for the AccessLevel can be measured by subtracting the end of InitialChecks from the start of ParseRequest.

Two query requests during the execution of the BIND uncached experiment, out of a total of thirty send to the server, a different/longer execution path was taken. BIND recursed and this multiplied the total performance time by a factor of 30 thousand. These measurements have been left out when the average execution time was calculated.

5.2.2 Measurement Results: Unbound

As with BIND, the measurement results for each measured check point is shown in Table 5.20. No anomalies were encountered when taking the measurements for Unbound. However, the measurement value for the function range IteratorProcessTargetResponse were not found inside the logs, even though these points have been executed. Of course, this affects the execution time calculated in the tables below, but the impact of missing this one function range does not have consequences when Unbound is compared to BIND.

[6]It must be noted however that the summed value does not represent the total execution time of one request. It only represents the total execution time if each range has been executed once

	cache run 1	cache run 2	cache run 3	Mean	CPU	Memory
InitialChecks	0.060022222222	0.058144444444	0.067422222222	0.061862962963	0.611979361	0.358638155
AccessLevel	2.432655555556	2.423788888889	2.536122222222	2.464188888889	24.37698858	14.28564225
ParseRequest	0.161033333333	0.159466666667	0.193300000000	0.171266666667	1.6942555	0.992884247
ClientAllowed	0.009844444444	0.010055555555	0.011111111111	0.010337037037	0.102259139	0.059926905
DestAddrAllowed	0.005411111111	0.005377777777	0.005411111111	0.005400000000	0.0534195	0.03130542
DnsViewAttach	0.010088888888	0.010066666666	0.012277777777	0.010811111110	0.106948917	0.062675254
SignatureChecks	0.021588888888	0.020266666666	0.023544444444	0.021799999999	0.2156565	0.12638114
RecursionAvailable	0.158233333333	0.152444444444	0.176411111111	0.162362962963	1.606175611	0.941266805
PrepareQStart	0.089022222222	0.089211111111	0.108344444444	0.095525925926	0.944990222	0.55379245
PrepareQFind	0.019766666666	0.018511111111	0.022466666666	0.020248148148	0.200304806	0.117384589
QueryGetDB	0.182500000000	0.172566666667	0.200077777778	0.185048148148	1.830588806	1.072779629
ZoneChecks	0.018788888888	0.016711111111	0.018933333333	0.018144444444	0.179493917	0.105188788
SetupSearch	0.049177777777	0.048888888888	0.068266666666	0.055444444444	0.548484167	0.321428078
DnsDBFind	0.273733333333	0.265055555556	0.297200000000	0.278662962963	2.756673361	1.615492795
RpzZones	0.007155555555	0.006688888888	0.006844444444	0.006896296296	0.068221611	0.039979899
AnswerFound	0.165522222222	0.155200000000	0.174055555556	0.164925925926	1.631529722	0.95612507
AddAuth	1.285011111111	1.278944444444	1.335600000000	1.299851851852	12.85878444	7.535631141
CleanUp	0.047255555555	0.047511111111	0.052088888888	0.048951851851	0.484256194	0.283788571
QuerySend	1.903277777778	1.852900000000	1.882233333333	1.879470370370	18.59266064	10.89585358
				9.49632962962	68.863670994	40.356164766
Total time	6.900088888882	6.791799999995	7.191711111106	6.961199999994	68.8636709999	40.3561647599

Table 5.18: BIND timings results with server caching allowed. All results are in $ms \times 10^{-2}$

5.3 BIND and Unbound measurements comparison

One difference is immediately noticeable between the two implementations[7]. The measurements show that Unbound is up to seven times faster than BIND when the answer is in their cache. These results support the claims[8][9] found in [17] that Unbound is faster than BIND. Because of the assumption that the energy consumption and execution time have a linear correlation, this also means that Unbound consumes considerably less energy than BIND when answering a request from cache. The tables also show that unlike BIND, Unbound does not take the same execution path for the cached and uncached answers. Unbound examines its list of previously cached requests before it decides to ask another server for an answer. However, BIND has this notion of 'databases' for all of its data-structures that hold information. These structures are then 'attached' and 'detached' from the search mechanism. This difference between BIND and Unbound, i.e. taking the same execution path or returning early, is most likely the difference seen between the performance of BIND and Unbound.

The uncached version of BIND seemed to only add a few extra function ranges compared to Unbound. The size of Table 5.18 and Table 5.19 are about the same length. However, the cached and uncached measurement tables of Unbound show an almost threefold increase in size. But, if the uncached tables of BIND and Unbound are taking at face value, then Unbound is undoubtedly the faster implementation. Which, again, validates the findings found in [17]. Nevertheless, as mentioned previously, the execution path found for Unbound inside its logs shows that an average of 125 function ranges have been executed.

[7] In this case the tables do represent the execution path taken and are thus correct. This is not the case for the uncached versions.

[8] http://info.menandmice.com/blog/bid/37244/10-Reasons-to-use-Unbound-DNS

[9] http://www.circleid.com/posts/unbound_vs_bind_open_source_dns/

	non-cache run 1	non-cache run 2	non-cache run 3	Mean	CPU (J)	Memory (J)
InitialChecks	0.067344444444	0.069211111111	0.072055555555	0.069537037037	0.687895139	0.403127065
AccessLevel	2.664822222222	2.665900000000	2.681488888889	2.670737037037	26.42026614	15.48306382
ParseRequest	0.477577777778	0.525922222222	0.507922222222	0.503807407407	4.983914778	2.920722683
ClientAllowed	0.012811111111	0.014077777777	0.013711111111	0.013533333333	0.1338785	0.078456793
DestAddrAllowed	0.005655555555	0.005555555555	0.005288888888	0.005499999999	0.05440875	0.03188515
DnsViewAttach	0.010355555555	0.009544444444	0.013166666666	0.011022222227	0.109037333	0.063899129
SignatureChecks	0.035555555555	0.031966666666	0.031511111111	0.033011111111	0.326562417	0.191375314
RecursionAvailable	0.186455555556	0.182777777778	0.222444444444	0.197225925926	1.951057472	1.14337786
PrepareQStart	0.216911111111	0.211377777778	0.232800000000	0.220362962963	2.179940611	1.277510205
PrepareQFind	0.118522222222	0.114911111111	0.112655555556	0.115362962963	1.141228111	0.668793705
QueryGetDB	0.213511111111	0.213088888889	0.212222222222	0.212940740741	2.106516278	1.234481356
ZoneChecks	0.114311111111	0.110877777778	0.115077777778	0.113422222222	1.122029333	0.657542649
SetupSearch	0.096688888888	0.085411111111	0.084988888888	0.089029629629	0.880725611	0.516131472
DnsDBFind	0.171344444444	0.181600000000	0.183422222222	0.536366666666	5.30600725	3.109478477
RpzZones	0.006911111111	0.007966666666	0.007144444444	0.007340740740	0.072618278	0.042556476
DnsDbAttach	0.012644444444	0.011722222222	0.015333333333	0.013233333333	0.13091075	0.076717603
DnsDbFind	0.222677777778	0.244755555556	0.236644444444	0.234692592593	2.321696472	1.360583367
DelegationChecks	0.005366666666	0.005033333333	0.004944444444	0.005114814814	0.050598306	0.029652116
Recurse	2.929222222222	2.905311111111	3.211122222222	3.015218518518	29.82804919	17.48012632
AnswerFound	0.438211111111	0.454922222222	0.450377777778	0.447837037037	4.430227889	2.596245655
AddAuth	1.550388888889	1.609022222222	1.543266666667	1.567559259259	15.50707997	9.087611294
CleanUp	0.085455555555	0.090088888888	0.087000000000	0.087514814814	0.865740306	0.507349636
QuerySend	312.1836666667	296.5907333333	303.2376777778	304.0040259259	3007.359826	1762.40254
				326.670488	3231.587803	1893.806820
Total time	318.6038555	303.066666	310.007088	304.004025	3007.359826	1762.402539

Table 5.19: BIND timings results with server non-caching. All results are in $ms \times 10^{-2}$

Summing the energy consumption for each of these functions reveals the actual execution time, and thus energy consumption of Unbound. The actual measured execution time and the consumed energy for Unbound is 2.0531 ms and 3689.034 Joules respectively. Comparing this to the consumption value found for BIND in Table 5.19, it can be concluded that Unbound also consumes less energy compared to BIND when answering an uncached query request.

	Mean (ms)($\times 10^{-2}$)	s_N (ms)($\times 10^{-9}$)	σ (ms)($\times 10^{-5}$)	CPU (J)	Memory (J)
UnboundTryLocalZoneAnswer	0.1422666666	0.066	0.08137703	0.01407373	0.008247625
UnboundAccessControl	0.0740666666	0.009	0.03091206	0.007327045	0.004293867
UnboundRBTSearch	0.0688888888	0.151	0.12296742	0.006814833	0.003993696
UnboundRequestHeader	0.0124777777	0.023	0.04825344	0.001234364	0.000723374
UnboundAnswerFromCache	0.1940111111	3.602	0.60023040	0.019192549	0.011247406
UnboundDNSChecks	0.5295222222	0.780	27943505	0.052382986	0.030697992
Total time with cache	1.0212333333	3.937	6.2751715	0.101025507	0.05920396

Table 5.20: Unbound timing results with server caching allowed

	run 1	run 2	run 3	Mean	CPU (J)	Memory (J)
UnboundAccessControl	0.07511	0.07440	0.07711	0.07554	0.74727945	0.437928042
UnboundRequestHeader	0.01893	0.01820	0.02235	0.0198267	0.19613563	0.114941328
UnboundDNSChecks	0.53925	0.54527	0.54458	0.5430333	5.37195692	3.14812695
UnboundTryLocalZoneAnswer	0.14924	0.15025	0.15021	0.1499	1.48288575	0.86901527
UnboundRBTSearch	0.03119	0.03053	0.03181	0.0311767	0.30841550	0.180740683
UnboundAnswerFromCache	0.19100	0.19100	0.19100	0.191	1.88946750	1.1072843
UnboundPrepNewRequest	0.01861	0.02051	0.02206	0.0203933	0.20174072	0.118226078
IteratorCheckEffort	0.00940	0.00747	0.00774	0.0082033	0.08115114	0.047556991
IteratorDNSCacheLookup	0.12935	0.18333	0.12819	0.1469567	1.45376915	0.851952077
IteratorQueryAdjustments	0.02557	0.02578	0.02651	0.025953	0.25674005	0.150457327
IteratorDNSCacheFindDelegation	0.64394	0.65155	0.64912	0.648203	6.41234817	3.757827252
IteratorPrimeRoots	1.30184	1.29907	1.30876	1.303223	12.8921335	7.555174698
IteratorUselessDelPoint	0.00973	0.01011	0.01052	0.01012	0.10011210	0.058668676
IteratorProcessInitRequest2	0.05331	0.05447	0.05259	0.0534567	0.52882040	0.309904527
IteratorProcessInitRequest3	0.01076	0.01056	0.01076	0.0106933	0.10578347	0.061992268
IteratorQueryTargetsChecks	0.37243	0.41857	0.37378	0.38826	3.84086205	2.250859698
IteratorServerSelection	0.28465	0.27649	0.27366	0.2782667	2.75275333	1.61319554
IteratorQueryNetworkTarget	2.56677	2.46420	2.45349	2.49482	24.6800068	14.46321999
IteratorDNSPreCheckServerResponse	0.05272	0.08718	0.05794	0.0659467	0.65237773	0.382312804
IteratorCleanUpAndCacheAnswer	0.65526	0.64599	0.65205	0.6511	6.44100675	3.77462203
IteratorPrepareRestart	1.59245	1.63489	1.58957	1.6056367	15.8837610	9.308357641
IteratorProcessFinished	0.34069	0.33836	0.33989	0.3396467	3.35995498	1.969033814
IteratorSendReply	166.72911	174.47736	148.60171	163.269393	1615.14244	946.5216346
UnboundCleanupRequest	0.09389	0.09532	0.09476	0.0946567	0.93639140	0.548753287
				172.425407	1705.71829	999.601786
Total time	175.8952	183.71086	157.67016	172.4254067	1705.71833	999.601810

Table 5.21: Unbound timings results with no cache. Results are all in $ms \times 10^{-2}$

Chapter 6

Comparison of Results

The previous chapter analyzed the measurements taken for the CPU and Memory of the server located at SEFlab for instructions such as addition, multiplication, conditionals, loops. The chapter also analyzed the measurements when sending a query request to BIND and Unbound. This chapter utilizes the results found in previously and inputs them in the ECAlogic component models for Implicit, BIND and Unbound. This is all discussed in the Section 6.1. This is followed by presenting another research conducted by HvA student Remy Briem in Section 6.2. Lastly, Section 6.3 discusses any difference found between the values in Section 6.1, the research discussed in Section 6.2 and the execution time measurement results found in Chapter 5.

6.1 ECAlogic results

ECAlogic only allows integers to be used as values for variables. However, the values determined in the previous chapter are floating point numbers and cannot be used in ECA or ECM. The values are thus multiplied by a factor of 10^{11} and 10^{9} for the energy and execution time to make them compatible with ECAlogic. The values calculated by ECAlogic are then divided by the same factors mentioned above. The time unit is in milliseconds and Joules is used as the energy unit. The values used for the BIND ECAlogic component models are seen in Table 5.18 and Table 5.19 for a cached and uncached query request to a BIND server. The values used for the Unbound server component models are seen in Table 5.20 and Table 5.21 for a cached and uncached query request.

An, as close as possible interpretation of the actual implementation, ECAlogic port of BIND and Unbound are found in Appendix D and Appendix E. The execution path for both BIND and Unbound have been retrieved from the logs as a result of using the measurement strategy described in Chapter 4. These execution paths have been used to determine the actual energy consumption of the path during the answering of a query request. These values are then compared to the energy consumption estimated by ECAlogic.

ECAlogic makes estimations of the models its been given. When ECAlogic encounters a conditional it takes the branch that exhibits the highest consumption of energy as its value, i.e. it calculates the consumption of the path that consumes the most energy. However, this is not desired if the energy consumption of both branches need to be determined. In the case of the DNS servers, both the cached and uncached execution path are given their own ECA source code and ECM component model. The approximated energy consumption for the cached and uncached execution paths must therefore be determined separately.

Table 5.18 and Table 5.19 show the check points encountered during the execution paths of the cached and uncached requests while analyzing the log files of BIND. The tables shows that BIND executes roughly the same path in the cached and uncached query request. The tables also show that the energy consumption and execution time differ slightly, the biggest difference is found in QuerySend, which is responsible for sending the requested information back to the client.

The execution paths for Unbound paint a different picture compared to that of BIND. The execution path of the cached version is extremely short (Table 5.20). The uncached execution path for Unbound

	Time	Energy (ECA)
BIND	$314498055680 + 3707189812N$	$49352495992 + 586196215N$
Implicit		$11829493 + 6828846N$
Memory		$18231121240 + 214073251N$
Component		$31109545259 + 365294118N$
Unbound	$840023588 + (163374820537 + 8099844949M)N$	$404545545 + (27106826129 + 1293373212M)N$
Implicit		$639732 + (5040213 + 32699351M)N$
Memory		$58580368 + (10013969808 + 466258051M)N$
Component		$345325445 + (17087816108 + 794415810M)N$

Table 6.1: ECALOGIC energy consumption estimates for BIND and Unbound for a cached and uncached query search.

almost more than triples the total *type* of check points encountered (Table 5.21). Additionally, Unbound not only more than triples the amount of check point types, the total executed check point increases from a total of six, in the cached execution path, to an average of 125 check points during the uncached execution path.

The symbolic upper bounds determined by ECALOGIC for BIND and Unbound are found in Table 6.1. The formulas correctly show that Unbound has the form of a quadratic line and BIND has a linear implementation. The difference however is in the constants within the formulas. These are higher for BIND than the are for Unbound. This suggests that the methods in Unbound are implemented more efficiently than those found in BIND.

The approximation results determined for BIND and Unbound are found in Table 6.2. Comparing the values found for the cached BIND execution path of Table 6.2 with those found in Table 5.18, it can be concluded that ECALOGIC successfully estimated the energy usage and the amount of time it would take to execute. The same can be concluded for the values estimated by ECALOGIC for Unbound after comparing the results of Table 6.2 and Table 5.20. Thus, from the measurements and ECALOGIC estimations it can be concluded that Unbound answers a cached query request almost seven times as fast compared to BIND. Furthermore, the measurements validate the results found by ECALOGIC.

The results for the uncached versions of the DNS implementations are also found in Table 6.2. The values are determined by filling in values for N and M. For BIND the value N = 1 is used. For Unbound the values N = 1 and M = 5 are used. These values are based upon the execution path found in the logs.

	Time		Energy (ECA)				
	ECA	(ms)	CPU	Memory	Component	Total ECA (10^9)	Total (J)
Cached							
BIND	6807379345	0.068074	6909478	394000405	672321412	1073231295	1.073231
Unbound	1024911218	0.010249	1853249	59203963	101025510	102878759	0.162083
Uncached							
BIND	318205245492	3.182053	18658339	18445194491	31474839377	49938692207	49.938692
Unbound	204714068870	2.047141	169176700	12403840431	21405220603	33978237734	33.978238

Table 6.2: ECALOGIC energy consumption estimates for BIND and Unbound for a cached and uncached query search.

The results presented in Table 6.2 are indeed an over approximation for BIND compared to the values seen in the previous chapter. The energy estimation of ECAlogic for the value of BIND does approximate the expected value. ECAlogic is meant to produce an upper bound on the energy consumption, therefore, a higher value is to be expected. Thus, the results of ECAlogic truly approximate the expected values.

The energy consumption approximated for Unbound, however, are less than those expected in the previous chapter. This difference can be explained as follows: the execution logs files show that Unbound performs a certain loop that cannot be determined by ECAlogic. ECAlogic cannot guess the path that is executed by the implementation, and therefore, a different path is used in order to determine the energy consumption. The same discrepancy is found for the execution time and, therefore, the explanation given above also applies.

6.2 SURFnet: Remy Bien

There are virtually no published research on the energy consumption of DNS server implementations. Therefore, SURFnet formulated an assignment that was to be carried out by a student. The purpose of the assignment was to analyze different DNS resolvers in order to gain insights on how the server consumes electric energy[1]. The research was carried out by Remy Bien, a former student at HvA. Three different resolvers were analyzed as part of Remy's research, namely Microsoft DNS, BIND and Unbound. The main research question that Remy set out to answer was: 'Which of the three DNS server implementation is the most energy efficient?'.

Remy answered the research question by measuring the energy consumption of each server when answering query requests. The same dataset is used for each DNS server in order to measure the energy consumed when answering a workload of DNS queries. The dataset used consisted of 1240548 real DNS queries sampled from a live DNS server on SURFnet's network. The total time of the dataset is 10 minutes and 20 seconds. The measurement methodology used by Remy is as followed: each measurement is divided into eight different 'slots' and each slot is measured for a total of ten minutes and 20 seconds. The first two slots are used to measure the idle state of the server. Slots three through eight are designated to measure the energy consumption when the DNS queries are sent to the server. Slot three is used for measuring the servers energy consumption with an empty cache. The other slots are used to measure the energy consumption when the cached is already filled with data.

The results found by Remy conclude that BIND is more energy efficient than Unbound, both on Windows and Linux[2]. The results show that, once all slots were averaged, BIND consumes 5.43 Watt per seconds and Unbound 5.77 Watt per seconds on CentOS. A server running BIND therefore saves the operator a total of €0.52 cents each year on energy costs. The results also showed that BIND consumes an extra 1.5 Watt per second if the dataset was not already cached. However, the energy consumption of Unbound remained nearly the same compared to the average. It only showed a slight increase in its energy consumption.

6.3 Result differences

The results found in Section 6.1 show that, given the assumptions made for this thesis, Unbound is faster and therefore more energy efficient than BIND. The result of ECAlogic and the measurements show that Unbound is, at the least, seven times more performant than BIND. However, the results of Remy show that BIND is found to be most energy efficient of the two implementations. Therefore, the question to be answered is: which result is the correct one?

First and foremost, one obvious reason for the difference between this thesis and Remy's results is that the assumption taken by this thesis is incorrect. Remy does not make any assumptions and the assumption made by this thesis is that the hardware, in this case the CPU and Memory, only has two states of energy consumption. The results found by Remy show that this must not be the case. However, its been shown that Unbound is faster than BIND in [17]. Migault et al. show that, in general, Unbound is

[1] http://www.sos.cs.ru.nl/applications/master/SURFnet_DNS_energy.pdf
[2] Technically, Windows DNS is found to be the most energy efficient on Windows

found to be faster by 67% for DNS requests without taking the query load into consideration. When the query load is taken into consideration, the research concludes that BIND is only able to handle 28% of the maximum query load that Unbound is able to handle. Unbound also comes on top when comparing network latency and the impact of cache hit rate. Not to mention that the measurements by this thesis also found that Unbound is faster than BIND. In the previous section it was mentioned that the dataset used in Remy's research consists of a fixed set of query requests in a fixed time period. Taking the results of the measurements from this thesis and those found in [17] into consideration, it stands to reason that the work performed by the servers are not equal. The energy consumption is taken for the same time period for both server, however, the amount of queries processed during this period is not the same. This means that the results do not take the energy consumption per query request into consideration. Which, of course, this thesis does.

The difference in the results may also be due to the configuration of the measurements. This thesis configured the operating system to only utilize one of the available four cores. Furthermore, both BIND and Unbound have been configured to only use one thread in order to serve client requests. Remy does not specify the thread configuration of the servers, therefore, it is assumed that the default configurations have been used. This, of course, is done because ECAlogic does not yet have the capabilities to model multi-core processors. The amount of threads used for BIND and Unbound is equal to the number of CPUs present on the system, thus four threads each[3][4]. Thus, in order to make the comparisons more equal to each other, either the measurement by Remy must be run using the same configurations as this thesis, or ECAlogic must be extended to support multiple cores and threads.

Lastly, the most important reason why these results do not math, is that Remy takes the energy measurement of the server as a whole. This thesis only looked at the energy consumption for the CPU and Memory hardware components. Remy, however, look at every component that was connected to the measurement setup. Furthermore, his thesis did not separate the energy consumption per hardware. Hence, other components that were also measured might influence that total energy consumed during the measurements.

[3]https://www.unbound.net/documentation/unbound.conf.html
[4]http://serverfault.com/questions/339742/how-to-configure-the-number-of-processes-bind-uses

Chapter 7

Discussion

This chapter discusses the measurement methodology presented in Chapter 4 and provides suggestions on how to improve the measurement methodology to acquire more accurate values. It continuous by discussing ECALOGIC and also suggests some improvements.

7.1 Measurements

The measurements methodology described in Chapter 4 seem to provide realistic values. The energy measurement performed for the `assignment` statement show that it took almost the same time to perform 320 million instructions as was determined in Table 5.1. The methodology used for measuring the statements needed for ECALOGIC used a similar method described in [28], and both seem to agree on the size of the body loop[1]. Furthermore, the execution times measured for `BIND` and `Unbound` do confirm the performance results found in [17][2]. The results of these different research give confidence in the measurement methodology used during this thesis. Also, the methodology used for `BIND` and `Unbound` provided a method to obtain the actual execution path for both servers. However, the approach taken to measure the energy consumption can still be improvement upon. Some improvements are provided below.

7.1.1 Context Switching

Chapter 4 presented the measurement methodology used for acquiring the energy consumption values. The techniques discussed is not perfect and improvement can be made. The values acquired during the measurement are not the most accurate that could have been collected. The hardware used for capturing the samples limits the amount of information that can be captured. The highest frequency possible for the hardware is `45KHz`. Ideally, the measurements would have been taken at 100KHz. As mentioned in Section 4.5, all the benchmarks are executed on one of the four cores available on the processor. Consequently, this signifies that the occurrences such as context switches are also effectively measured, because everything has to run on that one core. Thus, every value measured can truly be seen as an upper bound on the energy consumption and is not regarded as the actual energy consumed by the measurement methodology. In order to prevent the capturing of the context switches, its recommended to reserve one specific core on the processor. This strategy is mentioned in Section 4.5. Of course, the measurement hardware must be equipped to take energy measurements for each specific core.

7.1.2 Improve measurement implementation

Not only the methodology described in Chapter 4 can be improved upon, but also its implementation is not optimal. Both servers had to recurse in order to provide the query request with an answer. However, the implementation of these measurement points does not take this into account. Each time the server went into recursion, it overwrote the previous values of the check points. This is certainly a weak point of the energy consumption measurements measured for both `DNS` servers. In order to fix

[1] This thesis performed its own measurements to determined the size of the body loop.

[2] Of course, the confirmation is relative to the hardware used. But both research conclude that `Unbound` out performs `BIND` .

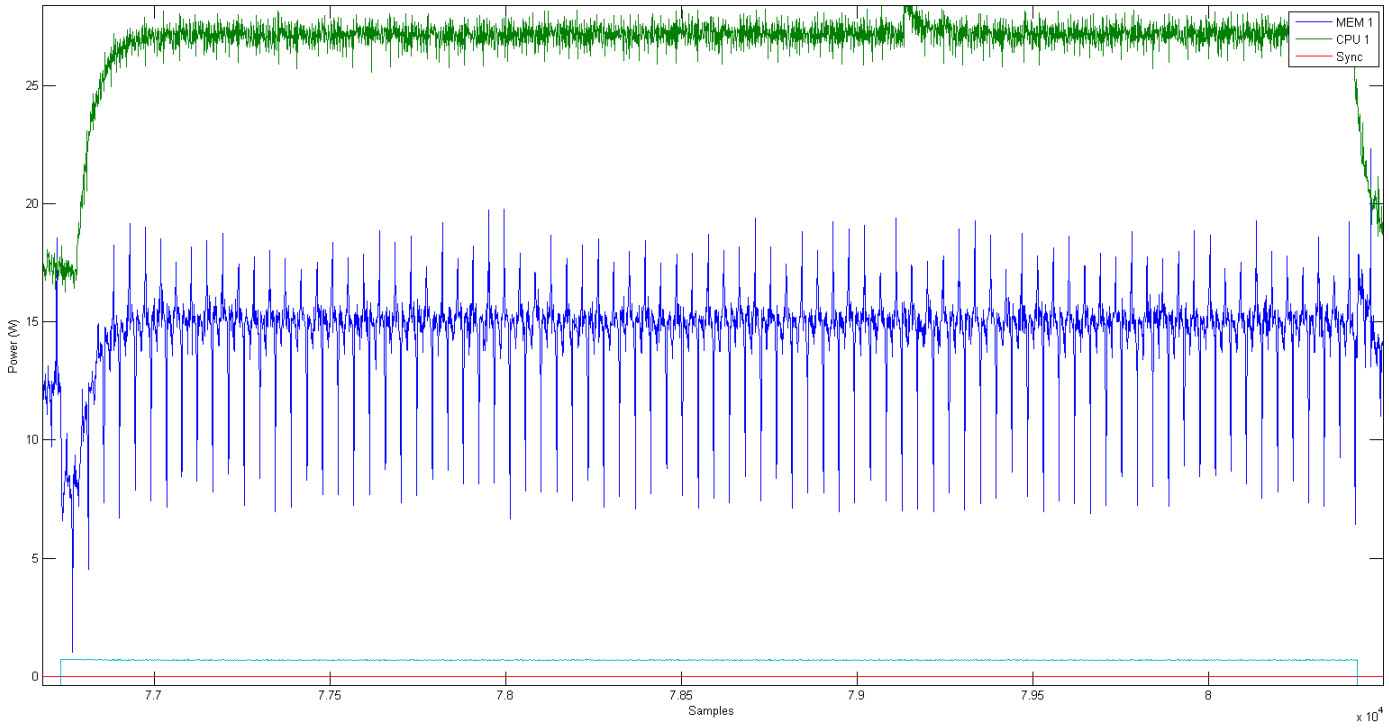

Figure 7.1: Assigning a value from one register to another one

this, an array that keeps track for each check point should be kept. This not only gives an execution path, which already has been achieved with current the methodology, but also keeps all the execution time of each individual encountered check point. This also improves the current method, because no time is needed to log the checkpoints to a file during the execution path. Which, of course, improves the performance measurement.

7.1.3 Hardware trailing power state

In their paper 'Fine-grained power modeling for smartphones using system call tracing' Pathak et al. mention that certain hardware components exhibit a behavior explained as the 'trailing power state phenomenon' [22]. This phenomenon is known as the tail power state. This state is when the hardware component does not switch immediately from a high energy consumption level to a level that demands less energy. Undoubtedly, such an occurrence can skew the energy consumption estimations that ECALOGIC makes. These estimations can then be lower than the actual energy consumption of the hardware in question. This can be mitigated by trying to find this trail power state for each hardware and add this as constant to the components ECM model that is used by ECALOGIC.

7.1.4 Implicit Memory

ECALOGIC has the implicit functions for the `CPU`, however, the measurements show that also the memory consumes energy implicitly when assigning from one `CPU` register to another. Figure 7.1 shows a measurement of the assembly instruction used for assigning a value from register to another register. Even though the `CPU` should not be retrieving values from memory, the figure shows that the energy consumption of the memory also increases during the measurement. One possible explanation is that the instruction pipeline is busy fetching instructions from memory. Another explanation is that context switches during the measurement are causing the constant energy consumption of the memory. This needs to be studied further, but it also provides an argument to also make the memory component an integral part of ECALOGIC.

7.1.5 Circuit switching

In their paper 'Instruction level power analysis and optimization of software' Tiwari et al. remarked that the switching activity of the circuit also consumes energy. Two different instructions cause a change in its state, and therefore, this changing of state causes the circuit to consume energy [28]. The same instructions are executed for `assignments` and `binary` statement in Chapter 5. Therefore, the change in state is significantly less during these measurements. This state change is captured for the other statements measured. The claims made by Tiwari et al. are confirmed in [6] by Chang et al. Furthermore, in their paper 'An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors' by Lee et al. state that the energy consumed in a clock cycle is dependent among other thing on the number of logical 1's [14]. The conclusions mentioned above are not taking into account in this thesis. Therefore, the energy consumption is not entirely accurate, but, in the spirit of ECALOGIC, are just upper bounds on the energy consumption.

7.2 ECALOGIC

This section discusses various pain points encountered while developing the examples of Section 2.4 and in porting the C code of the DNS server implementations over to ECA. Such pain points also include the experience working with the tool itself and its ease of use for an inexperienced user working with the tool.

ECALOGIC is definitely a novel idea that presents a very interesting approach to energy consumption analysis. Section 2.4 showed how simple algorithms can be implemented in ECA and analyzed for their energy consumption and Chapter 3 introduced the `DNS` servers. The models were for the `DNS` servers were implemented without too much trouble, in spite not having every concept available in `C`. Take for example structures, pointers and returning values from functions. All these issues can be worked around by simply abstracting them away. Structures and pointers are used to pass information around to different functions and return values are usually used to indicate error codes. Working around structures and pointers is achieved by just leaving them out of the ECA source. Return values can be 'simulated'. The function must then be used as the expression in an assignment statement and just assume the function returns a value. ECALOGIC does not throw an error and it does not influence the power consumption estimation. The only concept that could not be worked around were function pointers.

Getting used to ECALOGIC does take some time. Thinking of workarounds for every concept does get tiresome and is a task that should be automated in the future. Trying to write the same application in ECA should not be encouraged and the developer should accept that certain concepts cannot be modeled and should be abstracted over, or even left out. This of course influences the energy consumption, however, the tool should be used as an estimation tool to compare different implementations. It should be made clear that a accurate consumption of energy should not be expected.

7.2.1 Component models composition

One and important shortcoming of ECAlogic is that the tool does give the developer the possibility to compose the energy consumption of different component models. Meaning, the developers are prohibited of referencing the energy consumption of other component models inside model functions of a particular component model. Chapter 3/4/5 introduces the models for namely, the CPU (Implicit) and Main Memory (Memory). Section 3.2 discusses the component models of both `Unbound` and `BIND`. One advantage that ECALOGIC provides is being able to differentiate and give a detailed report which component model consumes the most energy. Such as feature can be improved upon. ECALOGIC should be able to show the number of times each function has been called, along with the total energy consumed by the function during the execution path of ECALOGIC.

The component models of both `Unbound` and `BIND` make use of functions that utilize both the CPU and Main Memory. However, ECALOGIC does not allow a developer to indicate that one function can affect two different component models. This forces the developer to be explicit in ECA with every statement. Take for example a function that utilizes three different component models. All three must be explicitly stated at each call site of the function. Another problem is that two of the three

```
import component Memory
component Implicit
    component function a (value) uses 5 energy 5 time
        Memory::store(value)
    end function

    component function e (lhs, rhs) uses 10 energy 10 time
        Memory::retrieve(lhs)
        Memory::retrieve(rhs)
    end function

    component function w    uses 25 energy 25 time
    component function ite  uses 25 energy 25 time
end component
```

Listing 7.1: Composable component

components must put there `time` field equal to zero. Otherwise ECALOGIC will triple its estimation on the execution time. One solution is to always assume that the energy consumption values assigned to the component model functions is the total energy consumption of all the hardware components it uses during the execution of that function[3].

7.2.2 Custom implicit component models

The limitations described in Section 7.2.1 have more implications than those mentioned in the section. Not since the days of assembly have programmers needed to reference memory directly in their own code. However, programmers of higher level programming languages work with memory implicitly. For example, accessing the data of variables in these languages implies touching the memory. Likewise, assigning data to a variable also means touching the systems memory. Therefore, every use of a variable and assignment operation carried out by the CPU influences the Memory component model[4].

This contradicts the assumptions taken by ECALOGIC, namely that component models do not influence each other and their energy consumption. Hence, in order to correctly estimate the energy consumption of saving and retrieving data from and to memory, the developer has to explicitly tell ECALOGIC that this energy consumption is taking place. This introduces extra complexity for developers using ECALOGIC that could be avoided. Take for example the `Insertion sort` algorithm presented in Listing 2.3 from Section 2.4. The ECA code references different variables in various places and also stores results of calculations in different variables. However, the monoInsertion sort example in ECA excludes the energy consumed by the memory. An updated version of the ECA code with all of the memory consumption annotations is seen in Listing 7.2. This new version adds an extra 13 lines of code to a simple and small example[5]. Certainly, the implicit nature of the energy consumption of the memory can be added to the $C_{implicit}$ component model as a constant. On the other hand, the developer then loses the fine granularity of insights that ECALOGIC can provide when its analysis is complete and it outputs the energy consumed by each component used by the ECA source code.

One solution is to allow developers to compose their component models and the energy they consume as discussed in Section 7.2.2. This solution is the most desirable, because it is simple and does not change the ECALOGIC implementation that much. Although, it is not immediately obvious from a theoretical point of view if such a solution is sound. One example of how such a feature implementation is presented in Listing 7.1. In this example, the `Implicit` component references C_{mem}. Therefore, the energy consumption is automatically included in the approximation by ECALOGIC. And more importantly, the store and retrieve component functions only have to be specified once, making the source code of ECA less verbose and the energy consumption estimations stay the same.

A second solution, albeit perhaps a less desirable one, is to allow developers to introduce their own

[3]This is less of an issue if the whole function is written inside ECM, however, this is not always possible. Sometimes you would want to abstract over the function, but also retain the ability to see precisely were it incurs energy consumption

[4]This is also seen in the energy consumption graphs from the previous chapters

[5]This examples is trying to be conservative and does not put each of the Memory component on its own line

```
import component Arr
import component Memory
function insertion_sort(A, N)
    i := 1
    Memory::store(i);
    Memory::retrieve(i); Memory::retrieve(N)
    while i < N bound N do
        Memory::retrieve(A); Memory::retrieve(i)
        value := Arr::get(A, i)
        Memory::store(value);  Memory::retrieve(i)
        j := i - 1
        Memory::store(j); Memory::retrieve(A); Memory::retrieve(j)
        jvalue := Arr::get(A, j)
        Memory::store(jvalue); Memory::retrieve(j);
        Memory::retrieve(jvalue);  Memory::retrieve(value);
        while j >= 0 and jvalue > value bound N do
            Memory::retrieve(A); Memory::retrieve(j);
            Arr::set(A, j + 1, j)
            Memory::retrieve(j);
            j := j - 1
            Memory::store(j);
        end while;
        Memory::retrieve(A);  Memory::retrieve(j)
        Arr::set(A, j + 1, value)
        Memory::retrieve(i);
        i := i + 1
        Memory::store(i);
    end while;
end function
function main(A, N)
    Memory::retrieve(A); Memory::retrieve(N)
    insertion_sort(A, N)
end function
```

Listing 7.2: Insertion sort updated to reflect memory energy consumption in ECA

custom implicit component models. These component models would then allow the developers to specify which statements and in which situations these custom component implicitly consume energy. ECALOGIC will then be able to add this energy consumption to its final estimation. How this feature should be implemented of look like is of this moment not known and has to be furthered studied.

7.2.3 Ports to ECALOGIC are essentially rewrites

The main limitation of ECALOGIC is that it requires developer to write their application in an entirely different language. This means that applications ported to ECA can essentially be considered as complete rewrites. Depending on the complexity of the application and the language concepts it utilizes, it might not be worth it to port the code to ECALOGIC. Porting the application requires the developers to maintain two different versions of the same application. Additionally, the developer need to be well versed in how to translate concepts to ECA. A more intelligent method for developers is to only model the path that is interesting for their energy consumption needs and keep the abstraction level high.

ECALOGIC has shown that it is easy enough to port simple algorithms to ECA. However, it becomes increasingly difficult to port code to ECA when they use concepts that are not supported. Take for example returning values from functions. The current version of ECALOGIC does not support this. Usually such a limitation can be worked around by just assuming that the function returned a value. Thus, the lack of certain programming language concepts can limit the accuracy in which an application can be ported to ECALOGIC. A more viable solution is to extend ECALOGIC to be able to analyze existing code bases. Furthermore, the limitation that not everything can be directly ported to ECALOGIC leave margin for errors in its estimation. This limitation is most likely imposed by the static analysis. Chapter 6 showed that the estimation for `Unbound` is less than the actual energy consumed. The reason for this is the dynamic nature of the application itself. Certain paths were taken that are difficult to trace without extra information.

7.2.4 Energy Analysis results of components

ECALOGIC in its current implementation can only show the total energy consumption of its component models. It does not provide an overview of the energy consumption at the function/statement level. Thus, ECALOGIC can only show the developer which component consumes the most energy. That in itself is interesting information to be aware about. However, developers are more likely to be interested in knowing which functions consume the most energy during the execution of their modeled path.

Take for example the `DNS` server analysis with ECALOGIC from Chapter 6. Table 6.2 show the results from ECALOGIC. It shows the energy consumption for the `CPU`, `BIND` and `Unbound` component model in `Joules`. Still, it does not show which functions are responsible for consuming the most energy. The results show that `Unbound` is more energy efficient than `BIND`. However, the developers of `BIND` have no insight on where exactly the energy is being consumed and which functions they have to improve to be more energy efficient. Of course, if the developers are taken their own measurements they already know. But, ECALOGIC should strive to relieve developers of taking their own measurements.

7.2.5 One file for each conditional

Every conditional requires ECALOGIC to make an estimate of the branch that consumes the most energy. workaround for this limitation is to not use the `else` branch in the source. The developer must introduce a variable to keep track of which branch must be taken. The downside to this is that the energy consumption estimation increases. In order to acquire the energy consumption of a specific execution path another solution must be considered. This involves creating a brand new ECA source file to obtain the energy estimations for that particular path. However, if the source has a lot of branches and each branch must have its own file, the number of files that need to be written grows exponentially. This, of course, is not what a developer would want. Theoretically, ECALOGIC should be able to generate energy estimations for all possible branch combinations. For small applications this could save the developer time. An alternative is to build a switch in that turns this feature on and off. A

Chapter 8

Conclusion and future work

This chapter concludes the thesis by summarizing the results found in the previous chapters and presents suggestions for future research on ECAlogic.

8.1 Conclusion

ECAlogic is a tool that, given a hardware and software model, is able to statically determine power consumption bounds for those particular models. This thesis set out to find the possibilities and limitations of ECAlogic by using it in a use case to determine the energy consumption of `DNS` server implementations.

This thesis looked into at which hardware components were needed to model the `DNS` servers in ECAlogic, introduced the ecm component models and suggested that its best to model the hardware components with an as simple as possible API. Hardware components for the `CPU`, `Memory` and `NIC` have been identified and with the right level of abstraction, all three were simple enough to construct their components in ECAlogic.

A measurement methodology has also been introduced and used to acquire the energy values for the component models. This thesis showed that taking measurements is not as easy as putting the same instruction a million times in the source code and measure the results. The best strategy found was to use a loop and execute a set number of statements within the body of the loop in order to minimize the effect of said loop. The theoretical limits of the processor were determined and the result for the simplest measurement test showed that the execution times fell close to each other, and thus serves as a small validation of the methodology. A method to acquire measurement values for the `DNS` servers have also been presented and successfully applied. The technique involved surrounding the functions of interest with logging methods that took the execution time and at the same time mapped the path taken by the server. Additionally, mistakes and improvements have been identified and solutions have been proposed.

ECAlogic makes it possible to determine the energy consumptions of simple algorithms and, this thesis has shown that, with a certain level of abstraction, complex `DNS` server implementations can also be modeled. Two algorithms were compared to each other and their energy consumption have been determined. Furthermore, implementations for `BIND` and `Unbound` have been modeled in ECAlogic and their energy consumption determined with real world energy consumption values. The results showed that the consumed energy for `BIND` have been correctly estimated and those of `Unbound` were underestimated. The results also showed that, given the assumption that the energy consumption and time are correlated, `BIND` makes the `CPU` and `Memory`, on average, consume more energy than `Unbound`. However, the results do not agree with the one found by `Remy Bien`. The difference can be attributed to the fact that this research only looked at the `CPU` and `Memory`, but `Remy` took the energy consumption of every component into consideration. Other factors such as the used number of cores and threads could have influenced the difference between the research projects. Furthermore, this thesis also made an assumption that may be invalid because of the results found by `Remy`. This thesis thus concludes that further research in which the energy consumption of the servers functions are also measured must

be performed. This project must therefore confirm whether the assumption made by this thesis is correct.

8.2 Future Work

Chapter 7 briefly mentioned that context switch occurrences can influence the energy consumption measured during the tests. In order to get execution time measurements that are more accurate, it might be interesting the determine how many context switches occur during the time it takes to run the application under test. `Linux` has the necessary tools that make it possible to determine how many context switches occur during the execution of an application. A method must be developed to discover how long a context switch takes, or literature on this topic has to be found. This time can then be subtracted from the total execution time of the application. Knowing ensures that a more accurate time is used for ECAlogic.

ECAlogic is not capable of analyzing existing code bases. Every application must be ported to ECA and ECM in order to determine its energy consumption. Therefore, it might be interesting to extend the capabilities of ECAlogic to, for example, analyze C applications. The project can use CLang[1][2] to acquire the Abstract Syntax Tree (AST) of C programs. This AST can then be utilized by ECAlogic to apply its energy consumption analysis. Naturally, this means that ECAlogic must be extended with new statements, such as pointers and structures (`struct`), that are available in C but not ECA. This is needed in order to make an accurate energy consumption estimation of C applications.

Developing best practices and a common style guide is essential for a any programming language[3]. Having a uniform manner of programming makes it easy for any developer of the language to quickly start contributing to a project. ECAlogic is no different. ECAlogic can develop best practices for porting, for example, code from the `C` programming language. This can help developers that use ECAlogic in the future in their journey of learning to work with ECAlogic. Examples of best practices include strategies such as converting function with return values to ECAlogic code. Another examples might include details on how to best deal with programming languages that include the possibility to jump to labels at specific points (such as 'GOTO' in `C` and 'break LABEL' in `Java`). Such style guides can then be used to develop an ECAlogic transpiler. This certainly can help newcomers to port their code to ECAlogic.

The last suggestion can be taken a step further. Once language port strategies have been determined, source code transpilers can be used to automate the translation of programming languages to ECAlogic. For example, the AST output of `CLang` can be used to automatically transpile the `C` programming language to ECAlogic. The best practices are thus automatically applied to the ECAlogic source. Effectively, minimizing the amount of effort that is needed to rewrite the ECAlogic source by hand. The generated code can then be used by the developers to build their own intuition on how to port source from other languages to ECAlogic, or better yet, write their own transpiler.

Finally, another power estimation method has been developed by Bircher et. al in their paper title 'Complete System Power Estimation using Processor Performance Events'. The authors are able to estimate that have an average error of less than 9%. This is done without the use of any additional sensors on the individual hardware components [3].

[1] Clang: a C language family frontend for LLVM
[2] http://clang.llvm.org/
[3] Think of C++ and the different parts that are not allowed by, for example, the 'Google C++ Style Guide'.

Bibliography

[1] Ron Aitchison and Joe Topjian. *Pro DNS and BIND 10*. Springer, 2011.

[2] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. *SIGPLAN Not.*, 45(6):198–209, June 2010. ISSN 0362-1340. doi: 10.1145/1809028.1806620. URL http://doi.acm.org/10.1145/1809028.1806620.

[3] W.L. Bircher and L.K. John. Complete system power estimation using processor performance events. *Computers, IEEE Transactions on*, 61(4):563–577, April 2012. ISSN 0018-9340. doi: 10.1109/TC.2011.47.

[4] C. Bunse, H. Hopfner, E. Mansour, and S. Roychoudhury. Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. In *Mobile Data Management: Systems, Services and Middleware, 2009. MDM '09. Tenth International Conference on*, pages 600–607, May 2009. doi: 10.1109/MDM.2009.103.

[5] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low power cmos digital design. *IEEE Journal of Solid State Circuits*, 27:473–484, 1995.

[6] Naehyuck Chang, Kwanho Kim, and Hyung Gyu Lee. Cycle-accurate energy consumption measurement and analysis: Case study of arm7tdmi. In *Low Power Electronics and Design, 2000. ISLPED'00. Proceedings of the 2000 International Symposium on*, pages 185–190. IEEE, 2000.

[7] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. *SIGPLAN Not.*, 47(10):831–850, October 2012. ISSN 0362-1340. doi: 10.1145/2398857.2384676. URL http://doi.acm.org/10.1145/2398857.2384676.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

[9] M.A. Ferreira, E. Hoekstra, B. Merkus, B. Visser, and J. Visser. Seflab: A lab for measuring software energy footprints. In *Green and Sustainable Software (GREENS), 2013 2nd International Workshop on*, pages 30–37, May 2013. doi: 10.1109/GREENS.2013.6606419.

[10] Gartner. Gartner says smartphone sales surpassed one billion units in 2014, 2015. URL http://www.gartner.com/newsroom/id/2996817.

[11] Lovic Gauthier and Tohru Ishihara. Compiler assisted energy reduction techniques for embedded multimedia processors. In *Proceedings of the 2nd APSIPA Annual Summit and Conference*, pages 27–36, 2010.

[12] Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Wu Ye. Influence of compiler optimizations on system power. In *Proceedings of the 37th Annual Design Automation Conference*, pages 304–307. ACM, 2000.

[13] Rody Kersten, Paolo Parisen Toldin, Bernard van Gastel, and Marko van Eekelen. A Hoare Logic for Energy Consumption Analysis. 2013.

[14] Sheayun Lee, Andreas Ermedahl, Sang Lyul Min, and Naehyuck Chang. An accurate instruction-level energy consumption model for embedded risc processors. *SIGPLAN Not.*, 36(8):1–10, August 2001. ISSN 0362-1340. doi: 10.1145/384196.384201. URL http://doi.acm.org/10.1145/384196.384201.

[15] Barry M Leiner, Vinton G Cerf, David D Clark, Robert E Kahn, Leonard Kleinrock, Daniel C Lynch, Jon Postel, Larry G Roberts, and Stephen Wolff. A brief history of the internet. *ACM SIGCOMM Computer Communication Review*, 39(5):22–31, 2009.

[16] Aqeel Mahesri and Vibhore Vardhan. Power consumption breakdown on a modern laptop. pages 165–180. 2005. doi: 10.1007/11574859_12. URL http://dx.doi.org/10.1007/11574859_12.

[17] Daniel Migault, Cédric Girard, and Maryline Laurent. A performance view on dnssec migration. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 469–474. IEEE, Oct 2010. doi: 10.1109/CNSM.2010.5691275.

[18] P. Mockapetris. DOMAIN NAMES - CONCEPTS AND FACILITIES. RFC 1034, Network Working Group, November 1987. URL http://tools.ietf.org/html/rfc1034.

[19] P. Mockapetris. DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION. RFC 1034, Network Working Group, November 1987. URL http://tools.ietf.org/html/rfc1034.

[20] James Pallister, Simon Hollis, and Jeremy Bennett. Identifying compiler options to minimise energy consumption for embedded platforms. *CoRR*, abs/1303.6485, 2013. URL http://arxiv.org/abs/1303.6485.

[21] Paolo Parisen Toldin, Rody Kersten, Bernard van Gastel, and Marko van Eekelen. Soundness Proof for a Hoare Logic for Energy Consumption Analysis. Technical Report ICIS–R13009, Radboud University Nijmegen, October 2013.

[22] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 153–168, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0634-8. doi: 10.1145/1966445.1966460. URL http://doi.acm.org/10.1145/1966445.1966460.

[23] Marc Schoolderman, Jascha Neutelings, Rody Kersten, and Marko van Eekelen. Ecalogic: Hardware-Parametric Energy-Consumption Analysis of Algorithms. January 2014. accepted for publications in the proceedings of the FOAL '14 workshop.

[24] Rathijit Sen and David A Wood. Cache power budgeting for performance.

[25] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2008. ISBN 0136006329, 9780136006329.

[26] Sanket Tavarageri and P Sadayappan. A compiler analysis to determine useful cache size for energy efficiency. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 923–930. IEEE, 2013.

[27] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, Dec 1994. ISSN 1063-8210. doi: 10.1109/92.335012.

[28] V. Tiwari, S. Malik, A. Wolfe, and M.T.-C. Lee. Instruction level power analysis and optimization of software. In *VLSI Design, 1996. Proceedings., Ninth International Conference on*, pages 326–328, Jan 1996. doi: 10.1109/ICVD.1996.489624.

[29] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for Reduced CPU Energy. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267638.1267640.

Appendices

Appendix A

Reserve a specific core on Linux

In Chapter 4.5 we explained that a Linux based operating system be configured to reserve and utilize a specific CPU. The following steps need to be carried to configure the operating system:

1. sudo -i
2. gedit /boot/grub/grub.conf
3. add line 'isolcpus=1' at the end of the appropriate kernel lines (i.e. kernel /boot/...)
4. save grub.conf
5. restart OS

The line 'isolcpus=1' configures the system scheduler to not assign any process or tasks to CPU 1. This unutilized CPU is then used to execute the applications. This is done by manually assigning an application to this CPU. This is done by entering one of the following command on the command line:

```
(1) taskset -c 1 ./{application}
(2) taskset -c 1 -p <pid>
```

The first command is used to run the application that measure expressions, assignments, conditionals and loop statements. The second is used to move our DNS server process with pid to CPU 1.

In order to validate that the CPU is not being utilized, the system will be used by opening a browser and visiting nu.nl. It is then verified that the CPU is not utilized by looking at its energy consumption[1].

[1] Ofcourse, the energy consumption of the operation also verified when the CPU has not been reserved

Appendix B

Implicit measurement setup

B.1 Directory structure

The directory structure for compiling and running the applications needed to measure the energy consumption of needed for the Implicit component model for ECAlogic.

```
root
├── bin
├── include
│   └── utils.h
├── results
├── src
│   ├── assignment.c
│   ├── binary.c
│   ├── function.c
│   ├── if_else.c
│   ├── .utils.c
│   └── while.c
├── build
└── measure
```

B.2 Timing utilities

The header

```c
/**
 * utils.h
 * This file contains global definitions used for measuring energy
    consumption.
 * It contains code for keeping the time and a helper function for logging
    files.
 */
#include <time.h>

typedef struct program_time {
    struct timespec start;
    struct timespec end;
} program_time;

/**
 * Mark our start time
 */
void start(struct program_time* time);

/**
 * Mark our end time
 */
void end(struct program_time* time);

```

```
/**
 * Opens a file and appends the run to the log file.

 * @param string        The name of the application
 * @param program_time  The time during this execution
 */
void logtime(char* program_name, struct program_time* time);
```

The source.

```
#include <time.h>
#include <stdio.h>

#include "utils.h"

void settime(struct timespec* time)
{
    clock_gettime(CLOCK_REALTIME, time);
}

void start(struct program_time* time)
{
    settime(&time->start);
}

void end(struct program_time* time)
{
    settime(&time->end);
}

void logtime(char* program_name, struct program_time* time)
{
    FILE* file = fopen(program_name, "ab+");

    if(file != NULL)
    {
        fprintf(file, "-----------------------------------------\n");
        fprintf(file, "start:\t%lld.%.9ld\n",
            (long long) time->start.tv_sec, time->start.tv_nsec);

        fprintf(file, "end:\t%lld.%.9ld\n",
            (long long) time->end.tv_sec, time->end.tv_nsec);

        fprintf(file, "-----------------------------------------\n");
        fclose(file);
    }

    else
    {
        printf("could not open file '%s' for writing...\n", program_name);
    }
}
```

B.3 Build

```
#!/bin/bash

echo "start build..."
START=$(date +%s.%N)
gcc -O0 ./src/$1.c ./src/utils.c -std=gnu99 -o ./bin/$1 -I"./include/" -lrt
    -Wall
END=$(date +%s.%N)
echo "build succeeded"
echo "build took: " $(echo "$END - $START" | bc) "seconds"
```

B.4 Measure

```
#!/bin/bash
/home/seflab/bin/seflabtools.sh -t synch -s /dev/ttyUSB0 -m run \
    -c /home/thesis/bin/$1 -o ./results/$1.server
```

Appendix C

Measurement Results

C.1 Assignment

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	13.31	6899.28	15.9933	3.8397	2.604
Memory	14.31	3764.18	9.4105	3.8397	2.604
All components					

Table C.1: 2^{nd} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	28.33	13617.00	17.0212	3.8464	2.60
Memory	29.78	7832.30	9.7903	3.8464	2.60
All components					

Table C.2: 3^{rd} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	58.89	28302.00	17.6888	3.8324	2.609
Memory	60.18	15828.26	9.89266	3.8324	2.609
All components					

Table C.3: 4^{th} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	120.49	57906.96	18.0959	3.8332	2.609
Memory	121.04	31837.72	9.9493	3.8332	2.609
All components					

Table C.4: 5^{th} run

C.2 Binary

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	87.93	38897.31	97.2432	22.5603	0.443
Memory	88.86	23114.22	57.7855	22.5603	0.443
All components					

Table C.5: 2^{nd} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	178.17	78816.94	98.5211	22.5407	0.444
Memory	179.40	46661.94	58.3274	22.5407	0.444
All components					

Table C.6: 3^{rd} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	359.15	158872.46	99.2952	22.5773	0.443
Memory	360.37	93734.26	58.5839	22.5773	0.443
All components					

Table C.7: 4^{th} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	719.64	318334.72	99.4796	22.5446	0.444
Memory	720.66	187445.40	58.5766	3.8332	22.5446
All components					

Table C.8: 5^{th} run

C.3 Function

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	63.4888	38940.91	97.3522	16.5223	0.605
Memory	65.31	17149.06	42.8726	16.5223	0.605
All components					

Table C.9: 2^{nd} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	131.64	80744.12	100.9301	16.7627	0.597
Memory	132.77	34864.12	43.5801	16.7627	0.597
All components					

Table C.10: 3^{rd} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	261.95	160670.44	100.4190	16.5200	0.605
Memory	263.31	15828.26	9.89266	16.5200	0.605
All components					

Table C.11: 4^{th} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	526.75	323085.52	100.9642	16.5347	0.603
Memory	528.22	138697.95	43.3431	16.5347	0.603
All components					

Table C.12: 5^{th} run

C.4 Conditional

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	110.46	47125.08	117.8127	28.2228	0.354
Memory	112.02	29019.35	72.5483	28.2228	0.354
All components					

Table C.13: 2^{nd} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	223.15	95198.16	118.9977	28.1505	0.355
Memory	224.64	58194.14	72.7426	28.1505	0.355
All components					

Table C.14: 3^{rd} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	448.57	191363.28	119.6020	28.1514	0.355
Memory	449.13	116347.99	72.7174	28.1514	0.355
All components					

Table C.15: 4^{th} run

C.5 While

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	104.77	45169.70	112.9242	26.7622	0.374
Memory	106.17	27569.06	68.9226	26.7622	0.374
All components					

Table C.16: 2^{nd} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	211.88	91345.30	114.1816	26.7592	0.374
Memory	213.62	55467.01	69.3337	26.7592	0.374
All components					

Table C.17: 3^{rd} run

	SEFlab (ms)	Total energy (J)(10^{-5})	Energy (J/Instr)	Time (ms/Instr)(10^{-7})	Processor (GHz)
CPU	426.26	183763.56	114.8522	26.7509	0.374
Memory	427.26	110939.79	69.3373	26.7509	0.374
All components					

Table C.18: 4^{th} run

Appendix D

Unbound

D.1 Server Configurations

The following configuration has been used when measuring the energy consumption of Unbound:

```
    server:
        interface: 192.168.10.11
        access-control: *laptop-ip* allow
        do-ip6: no
        root-hints: /etc/unbound/root.hints
        num-threads: 1 #no threading
```

D.2 Timing utility

The timing utility for Unbound looks like the following:

```
/**
 * utils.h
 * This file contains global definitions used for measuring energy
    consumption.
 * It contains code for keeping the time and a helper function for logging
    files.
 */
#include <time.h>

typedef struct program_time {
    struct timespec start;
    struct timespec end;
} program_time;

// worker.c
#define UnboundAccessControl                1
#define UnboundRequestHeader                2
#define UnboundDNSChecks                    3
#define UnboundTryLocalZoneAnswer           4
#define UnboundRBTSearch                    5
#define UnboundAnswerFromCache              6
#define UnboundPrepNewRequest               7

// iterator.c
#define IteratorCheckEffort                 9
#define IteratorDNSCacheLookup              10
#define IteratorResponseTypeFromCache       11
#define IteratorHandleCnameResponse         12
#define IteratorQueryAdjustments            13
#define IteratorDNSCacheFindDelegation      14
#define IteratorPrimeRoots                  15
#define IteratorUselessDelPoint             16
#define IteratorIsRoot                      17
#define IteratorLookupRoot                  18
```

```
#define IteratorDomainName                  19

#define IteratorProcessInitRequest2         20
#define IteratorProcessInitRequest3         21

#define IteratorQueryTargetsChecks          22
#define IteratorServerSelection             23
#define IteratorQueryNetworkTarget          24
#define IteratorDNSPreCheckServerResponse   25

#define IteratorCleanUpAndCacheAnswer       26
#define IteratorPrepareRestart              27
#define IteratorPrepareCnameReponseRestart  28

#define IteratorDNSStore                    29
#define IteratorProcessFinished             30

#define IteratorProcessTargetResponse       33

// mesh.c
#define UnboundCleanupRequest               31
#define IteratorSendReply                   32

#define START_TIMER(timer) start(&timers[timer])
#define END_TIMER(timer) end(&timers[timer])

#define LOGSTART(str, timer) log_info("START_TIMER \t:: ID %d :: \
    \t%lld.%.9ld", str, (long long) timer.start.tv_sec, timer.start.tv_nsec)
    ;

#define LOGEND(str, timer) log_info("END_TIMER \t:: ID %d :: \
    \t%lld.%.9ld", str, (long long) timer.end.tv_sec, timer.end.tv_nsec);

#define LOG_ECA(index) LOGSTART(index, timers[index]); LOGEND(index, timers[
    index]);

#define LOG_INFO(str) log_info("\n"); log_info(str);

#define LOGGER(str, index) LOG_INFO(str); START_TIMER(index);

/**
 * @brief timers
 * Global Timer array used by all modules inside Unbound
 */
program_time timers[33];

inline void settime(struct timespec* time) { clock_gettime(CLOCK_REALTIME,
    time); }

inline void end(struct program_time* time) { settime(&time->end); }
inline void start(struct program_time* time) { settime(&time->start); }
```

D.3 Compilation

We have to enable the GNU99 flag for the newest standard and link the real time library. Two lines have to be changed inside the makefile in order to make Unbound compile with our time utility library.

```
COMPILE=$(LIBTOOL) --tag=CC --mode=compile $(CC) -std=gnu99 $(CPPFLAGS) \
    $(CFLAGS)
LINK=$(LIBTOOL) --tag=CC --mode=link $(CC) $(staticexe) $(RUNTIME_PATH) \
    $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) -lrt
```

The added flags must be put exactly after $(CC) and $(LDFLAGS), otherwise the compiler will spew out error messages. Furthermore, two targets must be modified because they make use of the timing utility. The targets are `worker.lo` and `worker.o` and the following line has to be added to the list of dependencies:

```
$(srcdir)/util/timing/utils.h \
```

D.4 ECAlogic Evaluation Cached Component Model

```
component UnboundCache
    component function try_local_zone_answer()  uses 14073730 energy    142266667 time
    component function accesslevel()            uses 7327045 energy     74066667 time
    component function search()                 uses 6814834 energy     68888889 time
    component function request_header()         uses 1234365 energy     12477778 time
    component function answer_from_cache()      uses 19192550 energy    194011111 time
    component function dns_checks()             uses 52382986 energy    529522222 time
end component
```

D.5 ECAlogic Evaluation Cached Memory Component Model

```
component UnboundCacheMem
    component function try_local_zone_answer()  uses 8247626 energy     0 time
    component function accesslevel()            uses 4293867 energy     0 time
    component function search()                 uses 3993696 energy     0 time
    component function request_header()         uses 723375 energy      0 time
    component function answer_from_cache()      uses 11247407 energy    0 time
    component function dns_checks()             uses 30697992 energy    0 time
end component
```

D.6 ECAlogic Evaluation Cached ECA source

```
import component UnboundCache;
import component UnboundCacheMem;

function main()
    UnboundCache::accesslevel();
    UnboundCacheMem::accesslevel();
    UnboundCache::request_header();
    UnboundCacheMem::request_header();
    UnboundCache::dns_checks();
    UnboundCacheMem::dns_checks();
    UnboundCache::try_local_zone_answer();
    UnboundCacheMem::try_local_zone_answer();

    entry := UnboundCache::search();
    UnboundCacheMem::search();

    if entry <> 0 then
        answered := UnboundCache::answer_from_cache();
        UnboundCacheMem::answer_from_cache();
    else end if;
end function
```

D.7 ECAlogic Uncached Component Model

```
component Unbound
    component function accesslevel()            uses 7472795 energy 75540000 time
    component function request_header()         uses 264202661 energy 19826700 time
    component function dns_checks()             uses 53719569 energy 543033300 time
    component function try_local_zone_answer()  uses 14828858 energy 149900000 time
    component function search()             uses 3084155 energy 31176700 time
    component function prep_new_request()       uses 2017407 energy 20393300 time
    component function check_effort()           uses 811511 energy 8203300 time
    component function dns_cache_lookup()       uses 14537692 energy 146956700 time
    component function query_adjustments()      uses 2567401 energy 25953000 time
    component function dnscache_find_delegation()   uses 64123482 energy 648203000 time
    component function prime_roots()            uses 128921335 energy 1303223000 time
    component function process_initrequest2()   uses 5288204 energy 53456700 time
    component function process_initrequest3()   uses 1057835 energy 10693300 time
    component function iter_server_selection()  uses 27527533 energy 278266700 time
```

```
    component function query_checks()           uses 38408621 energy 388260000 time
    component function query_network_target()   uses 246800069 energy 2494820000 time
    component function dns_useless_del_point()  uses 1001121 energy 10120000 time
    component function dns_precheck_server_response()   uses 6523777 energy 65946700 time
    component function cleanup_andcache()       uses 64410068 energy 651100000 time
    component function prepare_restart()        uses 158837611 energy 1605636700 time
    component function process_finished()       uses 33599550 energy 339646700 time
    component function send_reply()             uses 16151424703 energy 163269393000 time
    component function cleanup_request()        uses 936391405 energy 94656700 time
    component function response_type_from_cache()   uses 0 energy 0 time
    end component
```

D.8 ECALOGIC Uncached Memory Component Model

```
component UnboundMem
    component function accesslevel()            uses 4379281 energy 0 time
    component function request_header()         uses 1149414 energy 0 time
    component function dns_checks()             uses 31481270 energy 0 time
    component function try_local_zone_answer()  uses 8690153 energy 0 time
    component function search()                 uses 1807407 energy 0 time
    component function prep_new_request()       uses 11072843 energy 0 time
    component function check_effort()           uses 1182261 energy 0 time
    component function dns_cache_lookup()       uses 8519521 energy 0 time
    component function query_adjustments()      uses 1504574 energy 0 time
    component function dnscache_find_delegation() uses 37578273 energy 0 time
    component function prime_roots()            uses 75551747 energy 0 time
    component function process_initrequest2()   uses 3099046 energy 0 time
    component function process_initrequest3()   uses 619923 energy 0 time
    component function iter_server_selection()  uses 16131956 energy 0 time
    component function query_checks()           uses 22508597 energy 0 time
    component function query_network_target()   uses 144632200 energy 0 time
    component function dns_useless_del_point()  uses 586687 energy 0 time
    component function dns_precheck_server_response() uses 3823129 energy 0 time
    component function cleanup_andcache()   uses 37746221 energy 0 time
    component function prepare_restart()    uses 93083577 energy 0 time
    component function process_finished()   uses 19690339 energy 0 time
    component function send_reply()         uses 9465216521 energy 0 time
    component function cleanup_request()    uses 548753287 energy 0 time
    component function response_type_from_cache()           uses 0 energy 0 time
end component
```

D.9 ECALOGIC Uncached ECA source

```
import component Unbound;
import component UnboundMem;

function handle_request(N, M)
    Unbound::accesslevel();
    UnboundMem::accesslevel();
    Unbound::request_header();
    UnboundMem::request_header();
    Unbound::dns_checks();
    UnboundMem::dns_checks();
    Unbound::try_local_zone_answer();
    UnboundMem::try_local_zone_answer();
    Unbound::search();
    UnboundMem::search();
    Unbound::prep_new_request();
    UnboundMem::prep_new_request();

    cont := 1;
    module_continue := 1;
    finished := 0;
    return_msg := 0;

    while module_continue <> 0 bound N do
        while module_continue = 1 and cont <> 0 bound M do
```

```
25              // init request state
26              if cont = 2 then
27                  Unbound::check_effort();
28                  UnboundMem::check_effort();
29                  msg := Unbound::dns_cache_lookup();
30                  UnboundMem::dns_cache_lookup();
31                  if msg <> 0 then
32                      Unbound::response_type_from_cache();
33                      UnboundMem::response_type_from_cache();
34                      cont := 8;
35                  else end if;
36
37                  // need to check if we are still in this branch
38                  if cont = 2 then
39                      Unbound::query_adjustments();
40                      UnboundMem::query_adjustments();
41                  else end if;
42
43                  inloop := 0;
44                  while cont = 2 and inloop <> 1 bound 1 do
45                      Unbound::dnscache_find_delegation();
46                      UnboundMem::dnscache_find_delegation();
47                      Unbound::dns_useless_del_point();
48                      UnboundMem::dns_useless_del_point();
49                      inloop := 1;
50                  end while;
51
52                  if cont = 2 then
53                      cont := 3;
54                  else end if;
55              else end if;
56
57              // init request 2 state
58              if cont = 3 then
59                  Unbound::process_initrequest2();
60                  UnboundMem::process_initrequest2();
61                  cont := 4;
62              else end if;
63
64              // init request 3  state
65              if cont = 4 then
66                  Unbound::process_initrequest3();
67                  UnboundMem::process_initrequest3();
68                  cont := 5;
69              else end if;
70
71              // query targets state
72              if cont = 5 then
73                  Unbound::query_checks();
74                  UnboundMem::query_checks();
75                  target := Unbound::iter_server_selection();
76                  UnboundMem::iter_server_selection();
77                  if target = 0 then
78                      cont := 0;
79                  else end if;
80                  if cont <> 0 then
81                      Unbound::query_network_target();
82                      UnboundMem::query_network_target();
83                      cont := 6;
84                  else end if;
85              else end if;
86
87              // query response state
88              if cont = 6 then
89                  result := Unbound::dns_precheck_server_response();
90                  result := UnboundMem::dns_precheck_server_response();
91                  if result = 1 then
92                      Unbound::cleanup_andcache();
93                      UnboundMem::cleanup_andcache();
94                  cont := 8;
95                  else end if;
96
97                  if  cont = 6 and result = 2 then
```

```
                    Unbound::prepare_restart();
                    UnboundMem::prepare_restart();
                cont := 5;
                else end if;
            else end if;

            // prime response state
            if cont = 7 then
                Unbound::prime_roots();
                UnboundMem::prime_roots();
            cont := 0;
            else end if;

            // process finished
            if cont = 8 then
                Unbound::process_finished();
                UnboundMem::process_finished();
                cont := 0;
                finished := 1;
            else end if;
        end while;

        if finished = 1 then
            if return_msg = 1 then
                Unbound::send_reply();
                UnboundMem::send_reply();
            else end if;
            Unbound::cleanup_request();
            UnboundMem::cleanup_request();
            module_continue := 0;
        else end if;
    end while;
end function

function main(N, M)
    handle_request(N, M);
end function
```

Appendix E

BIND

E.1 Server Configurations

The following configuration has been used when measuring the energy consumption of BIND:

```
options {
    directory "/var/named";
    listen-on { 127.0.0.1; };
    recursion yes;
    dnssec-enable no;
}

zone "." IN {
    type hint;
    file "named.root";
}

logging{
    channel simple_log {
        file "/home/thesis/thesis/logs/bind.log";
        severity debug;
        print-time yes;
        print-severity yes;
        print-category yes;
};

category default{
    simple_log;
    };
};

inlcude "/etc/rndc.key";

controls {
    inet 127.0.0.1 port 953
    allow { 127.0.0.1; } key { "rndc-key"; };
};
```

E.2 Timing utility

The timing utility for BIND looks like the following:

```
/**
 * utils.h
 * This file contains global definitions used for measuring energy
    consumption.
 * It contains code for keeping the time and a helper function for logging
    files.
 */
#ifndef UTILS_H
#define UTILS_H

#include <time.h>

typedef struct program_time {
    struct timespec start;
    struct timespec end;
} program_time;

// client.c
#define InitialChecks       1
#define AccessLevel         2
#define ParseRequest        3
#define Flags               4
#define ClientAllowed       5
#define DestAddrAllowed     6
#define DnsViewAttach       7
#define SignatureChecks     8
#define RecursionAvailable  9

#define PrepareQStart       10
#define PrepareQFind        11

#define QueryGetDB          12
#define ZoneChecks          13

#define SetupSearch         14
#define DnsDBFind           15
#define RpzZones            16

#define DnsDbAttach         17
#define DnsDbFind           18

#define DelegationChecks    19
#define RecursionOk         20
#define Recurse             21

#define CNamePrepareRestart 22
#define AnswerFound         23
#define AddAuth             24
#define CleanUp             25
#define QuerySend           26

#define START_TIMER(timer) start(&timers[timer])
#define END_TIMER(timer) end(&timers[timer])

#define bind_log(fmt, ...) isc_log_write(ns_g_lctx, NS_LOGCATEGORY_CLIENT,
    NS_LOGMODULE_CLIENT, \
            ISC_LOG_DEBUG(1), fmt, ##__VA_ARGS__);

#define LOGSTART(str, timer) bind_log("START_TIMER \t:: ID %d :: \
            \t%lld.%.9ld", str, (long long) timer.start.tv_sec, timer.
    start.tv_nsec);

#define LOGEND(str, timer) bind_log("END_TIMER \t:: ID %d :: \
            \t%lld.%.9ld", str, (long long) timer.end.tv_sec, timer.end.
    tv_nsec);

#define LOG_INFO_THESIS(str) bind_log("\n"); bind_log(str);
#define LOGGER(str, index) LOG_INFO_THESIS(str); START_TIMER(index);
#define LOG_ECA(index) LOGSTART(index, timers[index]); LOGEND(index, timers[
```

```
    index]);

/**
 * @brief timers
 * Global Timer array used by all modules inside Unbound
 */
program_time timers[33];

inline void settime(struct timespec* time);

inline void end(struct program_time* time);
inline void start(struct program_time* time);

#endif
```

E.3 Compilation

The build of BIND is a little bit more complicated than that of Unbound. Where Unbound only needed to change one file to include our timing utility, BIND requires changes in multiple files. The first modification must be added to the file located at `bind/bin/named/Makefile.in`. The list for `OBJS` and `SRCS` must be modified to include `utils.@O@` and `utils.c` respectively.

The second file that needs changing is located at `bind/make/rules.in`. As with the makefile for Unbound and our implicit applications, here too we add the GNU99 compiler flag and link to the real time library. The `FINALBUILDCMD` must be modified to the following:

```
FINALBUILDCMD = \
        if [ X"${MKSYMTBL_PROGRAM}" = X -o X"$${MAKE_SYMTABLE:-${ALWAYS_MAKE_SYMTABLE}}" =
     X ]; \
            then \
        ${LIBTOOL_MODE_LINK} ${PURIFY} ${CC} -std=gnu99 ${CFLAGS} ${LDFLAGS} -lrt \
        -o $@ $${BASEOBJS} $${LIBS0} ${LIBS}; \
    else \
        rm -f $@tmp0; \
        ${LIBTOOL_MODE_LINK} ${PURIFY} ${CC} -std=gnu99 ${CFLAGS} ${LDFLAGS} -lrt \
            -o $@tmp0 $${BASEOBJS} $${LIBS0} ${LIBS} || exit 1; \
        rm -f $@-symtbl.c $@-symtbl.@O@; \
        ${MKSYMTBL_PROGRAM} ${top_srcdir}/util/mksymtbl.pl \
            -o $@-symtbl.c $@tmp0 || exit 1; \
        $(MAKE) $@-symtbl.@O@ || exit 1; \
        rm -f $@tmp1; \
        ${LIBTOOL_MODE_LINK} ${PURIFY} ${CC} -std=gnu99 ${CFLAGS} ${LDFLAGS} -lrt \
            -o $@tmp1 $${BASEOBJS} $@-symtbl.@O@ $${LIBS0} ${NOSYMLIBS} || exit 1; \
        rm -f $@-symtbl.c $@-symtbl.@O@; \
        ${MKSYMTBL_PROGRAM} ${top_srcdir}/util/mksymtbl.pl \
            -o $@-symtbl.c $@tmp1 || exit 1; \
        $(MAKE) $@-symtbl.@O@ || exit 1; \
        ${LIBTOOL_MODE_LINK} ${PURIFY} ${CC} -std=gnu99 ${CFLAGS} ${LDFLAGS} -lrt \
            -o $@tmp2 $${BASEOBJS} $@-symtbl.@O@ $${LIBS0} ${NOSYMLIBS}; \
        ${MKSYMTBL_PROGRAM} ${top_srcdir}/util/mksymtbl.pl \
            -o $@-symtbl2.c $@tmp2; \
        count=0; \
        until diff $@-symtbl.c $@-symtbl2.c > /dev/null ; \
        do \
            count=`expr $$count + 1` ; \
            test $$count = 42 && exit 1 ; \
            rm -f $@-symtbl.c $@-symtbl.@O@; \
            ${MKSYMTBL_PROGRAM} ${top_srcdir}/util/mksymtbl.pl \
                -o $@-symtbl.c $@tmp2 || exit 1; \
            $(MAKE) $@-symtbl.@O@ || exit 1; \
            ${LIBTOOL_MODE_LINK} ${PURIFY} ${CC} -std=gnu99 ${CFLAGS} \
            ${LDFLAGS} -lrt -o $@tmp2 $${BASEOBJS} $@-symtbl.@O@ \
            $${LIBS0} ${NOSYMLIBS}; \
            ${MKSYMTBL_PROGRAM} ${top_srcdir}/util/mksymtbl.pl \
                -o $@-symtbl2.c $@tmp2; \
        done ; \
        mv $@tmp2 $@; \
        rm -f $@tmp0 $@tmp1 $@tmp2 $@-symtbl2.c; \
```

```
    fi
```

E.4 ECALOGIC Evaluation Cached Component Model

```
component BindCached
    component function initial_checks()         uses 6119794 energy 61862963    time
    component function access_level()           uses 243769886 energy 2464188889    time
    component function parse_request()          uses 16942555 energy 171266667  time
    component function clientAllowed()          uses 1022591 energy     10337037    time
    component function destinationsAllowed()    uses 534195 energy  5400000     time
    component function dns_view_attach()        uses 1069489 energy     10811111    time
    component function signature_checks()       uses 2156565 energy     21800000    time
    component function recursion_available()    uses 16061756 energy    162362963   time
    component function prepare_qstart()         uses 9449902 energy     95525926    time
    component function prepare_qfind()          uses 2003048 energy     20248148    time
    component function query_getdb()            uses 18305888 energy    185048148   time
    component function zone_checks()            uses 1794939 energy 18144444    time
    component function setup_search()           uses 5484842 energy 55444444    time
    component function dnsdb_find()             uses 27566734 energy 278662963  time
    component function rpz_zones()              uses 682216 energy 6896296      time
    component function answer_found()           uses 16315297 energy 164925926      time
    component function addauth()                uses 128587844 energy 1299851852    time
    component function cleanup()                uses 4842562 energy 48951852    time
    component function query_send()             uses 185926606 energy 1879470370    time
end component
```

E.5 ECALOGIC Evaluation Cached Memory Component Model

```
component BindCachedMem
    component function initial_checks()         uses 3586382 energy 0 time
    component function access_level()           uses 142856423 energy 0 time
    component function parse_request()          uses 9928843 energy 0 time
    component function clientAllowed()          uses 599270 energy  0  time
    component function destinationsAllowed()    uses 313055 energy  0 time
    component function dns_view_attach()        uses 626753 energy  0  time
    component function signature_checks()       uses 1263812 energy 0  time
    component function recursion_available()    uses 9412669 energy 0 time
    component function prepare_qstart()         uses 5537925 energy 0  time
    component function prepare_qfind()          uses 1173846 energy 0  time
    component function query_getdb()            uses 10727797 energy 0 time
    component function zone_checks()            uses 1051888 energy 0 time
    component function setup_search()           uses 3214281 energy 0 time
    component function dnsdb_find()             uses 16154928 energy 0 time
    component function rpz_zones()              uses 399799 energy 0 time
    component function answer_found()           uses 9561251 energy 0 time
    component function addauth()                uses 75356312 energy 0 time
    component function cleanup()                uses 2837886 energy 0 time
    component function query_send()             uses 108958536 energy 0 time
end component
```

E.6 ECALOGIC Evaluation Cached ECA source

```
import component BindCached
import component BindCachedMem

function iterate_viewlist(request, length)
    index := length;
    while index <> 0 bound length do
        client_allowed := BindCached::clientAllowed();
        BindCachedMem::clientAllowed();
        destaddr_allowed := BindCached::destinationsAllowed();
        BindCachedMem::destinationsAllowed();

        if client_allowed = 1 and destaddr_allowed = 1 then
```

```
            BindCached::dns_view_attach();
            BindCachedMem::dns_view_attach();
        else
            index := index - 1
        end if;
    end while;
end function

function bind(request, listlength)
    BindCached::initial_checks();
    BindCachedMem::initial_checks();
    BindCached::access_level();
    BindCachedMem::access_level();
    BindCached::parse_request();
    BindCachedMem::parse_request();

    iterate_viewlist(request, 1)

    BindCached::signature_checks();
    BindCachedMem::signature_checks();

    recursing := 1;
    BindCached::recursion_available();
    BindCachedMem::recursion_available();
    BindCached::prepare_qstart();
    BindCachedMem::prepare_qstart();
    BindCached::prepare_qfind();
    BindCachedMem::prepare_qfind();
    db := BindCached::query_getdb();
    BindCachedMem::query_getdb();

    BindCached::zone_checks();
    BindCachedMem::zone_checks();
    BindCached::setup_search();
    BindCachedMem::setup_search();
    result := BindCached::dnsdb_find();
    BindCachedMem::dnsdb_find();
    BindCached::rpz_zones();
    BindCachedMem::rpz_zones();

    BindCached::addauth();
    BindCachedMem::addauth();
    BindCached::cleanup();
    BindCachedMem::cleanup();
    BindCached::query_send();
    BindCachedMem::query_send();
end function

function main(request, N)
    bind(request, N);
end function
```

E.7 ECALOGIC Evaluation UnCached Component Model

```
component Bind
    component function initial_checks()         uses 687895139      energy 69537037 time
    component function access_level()           uses 26420266139    energy 2670737037 time
    component function parse_request()          uses 4983914778     energy 503807407 time
    component function clientAllowed()          uses 133878500      energy 13533333 time
    component function destinationsAllowed()    uses 54408750       energy 5500000 time
    component function dns_view_attach()        uses 109037333      energy 11022222 time
    component function signature_checks()       uses 326562417      energy 33011111 time
    component function recursion_available()    uses 1951057472     energy 197225926 time
    component function prepare_qstart()         uses 2179940611     energy 220362963 time
    component function prepare_qfind()          uses 1141228111     energy 115362963 time
    component function query_getdb()            uses 2106516278     energy 212940741 time
    component function zone_checks()            uses 1122029333     energy 113422222 time
    component function setup_search()           uses 880725611      energy 89029630 time
    component function dnsdb_find()             uses 5306007250     energy 536366667 time
    component function rpz_zones()              uses 72618278       energy 7340741 time
```

```
    component function dnsdb_attach()          uses 130910750      energy 13233333 time
    component function dnsdb_find2()           uses 2321696472     energy 234692593 time
    component function delegation_checks()     uses 50598306       energy 5114815 time
    component function recurse()               uses 29828049194    energy 3015218519 time
    component function answer_found()          uses 4430227889     energy 447837037 time
    component function addauth()               uses 15507079972    energy 1567559259 time
    component function cleanup()               uses 865740306      energy 87514814 time
    component function query_send()            uses 3007360000 energy 304004025900 time
end component
```

E.8 ECALOGIC Evaluation Uncached Memory Component Model

```
component BindMem
    component function initial_checks()        uses 4031271 energy 0   time
    component function access_level()          uses 154830639 energy 0 time
    component function parse_request()         uses 29207227 energy 0  time
    component function clientAllowed()         uses 784568 energy 0    time
    component function destinationsAllowed()   uses 318852 energy 0    time
    component function dns_view_attach()       uses 638992 energy 0    time
    component function signature_checks()      uses 1913754 energy 0   time
    component function recursion_available()   uses 11433779 energy 0  time
    component function prepare_qstart()        uses 12775103 energy 0  time
    component function prepare_qfind()         uses 6687938 energy 0   time
    component function query_getdb()           uses 12344814 energy 0  time
    component function zone_checks()           uses 6575427 energy 0   time
    component function setup_search()          uses 5161315 energy 0   time
    component function dnsdb_find()            uses 31094785 energy 0  time
    component function rpz_zones()             uses 425565 energy 0    time
    component function dnsdb_attach()          uses 767177 energy 0    time
    component function dnsdb_find2()           uses 13605834 energy 0  time
    component function delegation_checks()     uses 296522 energy 0    time
    component function recurse()               uses 174801264 energy 0 time
    component function answer_found()          uses 25962457 energy 0  time
    component function addauth()               uses 90876113 energy 0  time
    component function cleanup()               uses 5073497 energy 0   time
    component function query_send()            uses 17624025396 energy 0 time
end component
```

E.9 ECALOGIC Evaluation UnCached ECA source

```
import component Bind
import component BindMem

function iterate_viewlist(request, length)
    index := length;
    while index <> 0 bound length do
        client_allowed := Bind::clientAllowed();
        BindMem::clientAllowed();
        destaddr_allowed := Bind::destinationsAllowed();
        BindMem::destinationsAllowed();

        if client_allowed = 1 and destaddr_allowed = 1 then
            Bind::dns_view_attach();
            BindMem::dns_view_attach();
        else
            index := index - 1
        end if;
    end while;
end function

function bind(request, N)
    Bind::initial_checks();
    BindMem::initial_checks();
    Bind::access_level();
    BindMem::access_level();
    Bind::parse_request();
    BindMem::parse_request();
```

```
28
29      iterate_viewlist(request, 1)
30
31      Bind::signature_checks();
32      BindMem::signature_checks();
33
34      recursing := 1;
35      Bind::recursion_available();
36      BindMem::recursion_available();
37      Bind::prepare_qstart();
38      BindMem::prepare_qstart();
39      Bind::prepare_qfind();
40      BindMem::prepare_qfind();
41      db := Bind::query_getdb();
42      BindMem::query_getdb();
43
44      Bind::zone_checks();
45      BindMem::zone_checks();
46      Bind::setup_search();
47      BindMem::setup_search();
48      result := Bind::dnsdb_find();
49      BindMem::dnsdb_find();
50      Bind::rpz_zones();
51      BindMem::rpz_zones();
52
53      // not found
54      if result = 1 then
55          Bind::dnsdb_attach();
56          BindMem::dnsdb_attach();
57          Bind::dnsdb_find();
58          BindMem::dnsdb_find();
59          result := 2;
60      else
61      end if;
62
63      // delegation
64      if result = 2 then
65          Bind::delegation_checks();
66          BindMem::delegation_checks();
67          Bind::recurse();
68          BindMem::recurse();
69          recursing := 1;
70      else
71      end if;
72
73      while recursing <> 0 and result <> 5 bound N do
74          Bind::prepare_qfind();
75          BindMem::prepare_qfind();
76          Bind::rpz_zones();
77          BindMem::rpz_zones();
78
79          // not found
80          if result = 1 then
81              Bind::dnsdb_attach();
82              BindMem::dnsdb_attach();
83              Bind::dnsdb_find();
84              BindMem::dnsdb_find();
85              result := 2;
86          else
87          end if;
88
89          // delegation
90          if result = 2 then
91              Bind::delegation_checks();
92              BindMem::delegation_checks();
93              Bind::recurse();
94              BindMem::recurse();
95              recursing := 1;
96          else
97              // assume its been found
98              result := 5;
99          end if;
100     end while
```

```
101
102     // answer found
103     if result = 5 then
104         Bind::answer_found();
105         BindMem::answer_found();
106     else
107     end if;
108
109     Bind::addauth();
110     BindMem::addauth();
111     Bind::cleanup();
112     BindMem::cleanup();
113     Bind::query_send();
114     BindMem::query_send();
115 end function
116
117 function main(request, N)
118     bind(request, N);
119 end function
```