Ville Vainio

Memory optimization techniques in xStorage Compact

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Bachelor's Thesis

31 August 2021

Abstract

Author:	Ville Vainio
Title:	Memory Optimization techniques in xStorage Compact
Number of Pages:	31 pages
Date:	31 August 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Smart Systems
Supervisors:	Sami Sainio, Lecturer Ville Pirttinokka, xStorage Firmware technical leader

---

Real-time embedded systems are categorized on hard- and soft real-time systems based on if they use preemptive or non-preemptive priority scheduling and if they need to meet their performance objectives all the time or on average. xStorage Compact energy storage falls into the category of hard real-time embedded systems due to its need for critical matching of response times in environments where power supply simply cannot fail, for example in hospitals. For working with a hard-real-time system, the system needs to ensure that both the criteria of performance and size of the program match specifications.

This thesis focuses on the xStorage Compact energy storage PCS firmware side where there is a constant need for memory optimization due to the use of low memory constraint microcontrollers. The focus will be on the ECM side of the PCS due to need for memory optimization being the highest in that side of the firmware.

Due to the hard-real-time requirement of this system, code is rather added to the source code than refactored because that way it is easier to control the functionality of the new compiled code. This results into memory running out on the hardware in question overtime and is the basis why this thesis is created.

The starting point of this thesis is analyzing the C++ classes as objects implemented in the firmware from a memory consumption perspective, for example rectifier class consumes most of the memory according to memory map so it will be analyzed for optimization first. The idea is to look at the implementation and look for sections which could be refactored for memory optimization, structures and data members that are not handled and arranged by data type sizes, minimizing the amount of data structures and abstraction layers, looking at possible areas to implement dynamic memory allocation and minimizing memory overhead.

In this thesis the techniques implemented result into 1,5% of improvement in the memory consumption of RAM usage and 0,5% of the usage in flash memory. RAM memory is the more expensive one of the two and the results are reflecting the emphasis on this section of memory.

Abstract

Tekijä:	Ville Vainio
Otsikko:	Memory Optimization techniques in xStorage Compact
Sivumäärä:	31 Sivua
Aika:	31.8.2021
Tutkinto:	Insinööri (AMK)
Tutkinto ohjelma:	Informaatio ja kommunikaatioteknologia
Ammatillinen pääaine:	Älykkäät laitteet
Ohjaajat:	Sami Sainio, Lehtori Ville Pirttinokka, xStorage-laiteohjelmiston tekninen johtaja

---

Reaaliaikaiset sulautetut järjestelmät jaotellaan koviin ja pehmeisiin järjestelmiin riippuen siitä, käyttävätkö ne ennaltaehkäisevää vaiko ei-ennaltaehkäisevää prioriteettiaikataulua ja tarvitseeko niiden suorituskyvyn olla aina tiettyjen rajojen sisällä. xStorage Compact -energiavarasto kategorisoidaan kovaksi reaaliaikaiseksi järjestelmäksi, koska sen pitää ylläpitää kriittisiä reagointiaikoja ympäristöissä, joissa voiman syöttö ei voi katketa kuten sairaaloissa.
Koska laite on kova reaaliaikainen systeemi, suorituskyvyn ja ohjelmiston kokojen pitää olla määritelmien mukaiset. Tämä insinöörityö keskittyy xStorage Compact -energiavaraston PCS puolen laiteohjelmistoon, missä jatkuva tarve muistin optimoinnille on tarpeellinen, sillä mikrokontrollerissa on vähän muistia. Painotus työssä on ECM-puolelle, sillä siellä on korkein tarve muistin optimoinnille.
Koska laitteella on niin kriittiset vaatimukset, koodia mieluummin lisätään lähdekoodiin kuin kirjoitetaan uudelleen, sillä silloin on helpompi ylläpitää toimivuutta. Tästä syystä vuosien varrella muisti alkaa käydä vähiin ja lopputyön aihe tulee tarpeelliseksi.
Aloituskohta tälle työlle on C++-luokkien analysointi muistin kulutuksen perspektiivistä. Esimerkiksi tasasuuntaajaan käytetty luokka kuluttaa eniten muistia muistikartan mukaan, joten se on analysoinnin lähtökohta.
Ideana on tutkia implementaatiota ja katsoa koodista kohtia, joita voi kirjoittaa uudelleen muistin kannalta paremmin, rakenteita ja tietojäseniä, joita ei ole käsitelty ja lajiteltu tyypin koon mukaan, minimoida datarakenteiden määrää, katsoa mahdollisia kohtia implementoida dynaamista muistinhallintaa sekä minimoida muistikuluja.
Tässä työssä käytetyt tekniikat saavat aikaan 1,5 % parannuksen RAM-muistissa ja 0,5 % parannuksen Flash-muistissa. RAM-muisti on kalliimpaa käyttää, joten tulokset näyttävät, että siihen on keskitytty enemmän.

Contents

List of abbreviations

BIOS:	Basic input-output system.
COFF:	Common object file format.
DSP:	Digital signal processor.
EABI:	Embedded application binary interface.
ECM:	Energy conversion module, converts alternating current to direct current and vice versa.
EMEA:	Europe, the Middle East and Africa.
HMI:	Human machine interface.
HW:	Hardware.
IDE:	Integrated development environment
MCU:	Machine Control Unit.
PCS:	Power conversion system, controls the inverter, rectifier and all features that are a part of UPS.
RAM:	Random access memory.
UPS:	Uninterruptible Power Supply, is a device that provides emergency power to a load.
UPSaaR:	UPS-as-a-Reserve, or energy aware is a technology that allows UPS to give energy back to the grid.

List of concepts

Application controller:	Works as the user interface display in the xStorage Compact.
Embedded system:	Microcontroller based mechanical or electrical device that has logic on how it operates.
Flash:	Non-volatile memory that can keep data without power.
Microcontroller:	Compact and small computer chip that runs specific functionalities for an embedded system.
System controller:	Sends the commands between the PCS and application controller and works as a control and monitoring solution with web server that runs the web user interface.

1 Introduction

Microcontroller revolution which happened in 1980s brought back the constant need for memory and performance efficiency in embedded devices [1]. Due to the nature of how most of the embedded systems are built considering cost, the memory is restrained when using microcontrollers for logic.

Memory is constrained to match optimized performance, timing, power, and cost. It is hard to specify memory and performance requirements in contrast to specifications in a large-scale project. This means that programmers might have to compromise on some of the requirements or try to find different approaches to fit the program to specifications required.

When starting a project and instantly trying to optimize on-the-go will be almost impossible and counterproductive. Finding major performance- and memory consuming bottlenecks before the program is working completely is not practical. Steve McConnell in his book “Code complete 2nd edition” states that programmers tend to ignore significant global optimizations when doing a lot of micro-optimizations which is easy to happen when addressing issues on a not finished project. [1]

Large-scale firmware projects become hard to maintain due to changing environment and different programmers with different perspectives working on it. This is a big issue when it comes to critical real-time system functionalities which requires that the system cannot fail at any time.

It is much safer to add code to the project than to start refactoring a whole section. This way the risk is minimised, and the system is more guaranteed to work after code changes and easier to scope the areas that need to be retested. This results into more code that adds more performance and memory consumption after time.

This thesis aims to provide techniques for a strategy which is effective to follow when addressing issues regarding memory limitations in large scale projects

and manually addressing them. This means a solution to fine-tune the existing code to save memory usage for new features without constant refactoring and to provide an existing example of a strategy that has been done in the xStorage Compact firmware.

1.1 Eaton

Eaton Corporation is a global power management company founded in the United States by Joseph O. Eaton in 1911. [2] Eaton has two main business sectors which are electrical sector and industrial sector. Electrical sector is divided to America and EMEA regions and produces critical power management solutions. Industrial sector consists of hydraulics, aerospace, and vehicle.

Eaton research and development center located in Le Lieu, Switzerland is the site where xStorage Compact energy storage solution came into fruition and has majority of the development of this product conducted together with Eaton power quality. Eaton power quality is the subsidiary of Eaton Corporation based in Finland that produces and develops UPS systems.

1.2 UPS

Uninterruptible power supply is a device that provides emergency power to a load when input power or mains power fails or has conditions that would normally drop the load. UPS has three main functionality which are to provide clean power that has been conditioned from incoming dirty power, ride-through power for temporary outages and to have a smooth shutdown for system when there is a complete power outage longer than the UPS batteries can support the system.

Phases of UPS devices come in single-phase or three-phase defining the number of electrical phases that it receives and transmits. Large power consumption facilities are often three-phase since it is the most efficient way of transporting electricity.

1.3 xStorage Compact energy storage

Unlike other UPS devices the xStorage Compact, including normal UPS operation, is also an energy storage which means a system that by definition stores energy to be used later. xStorage Compact has a lot of advanced technologies that combine the UPS functionalities to energy storage features when processing power from the grid.

Figure 1 displays how the xStorage Compact is used as a part of an electric vehicle charging infrastructure. The system has a lot of unique and eco- and cost friendly features like its peak shaving that keep the load even and UPS-as-a-reserve technology which help the system be utilized as an investment by returning energy to the grid.

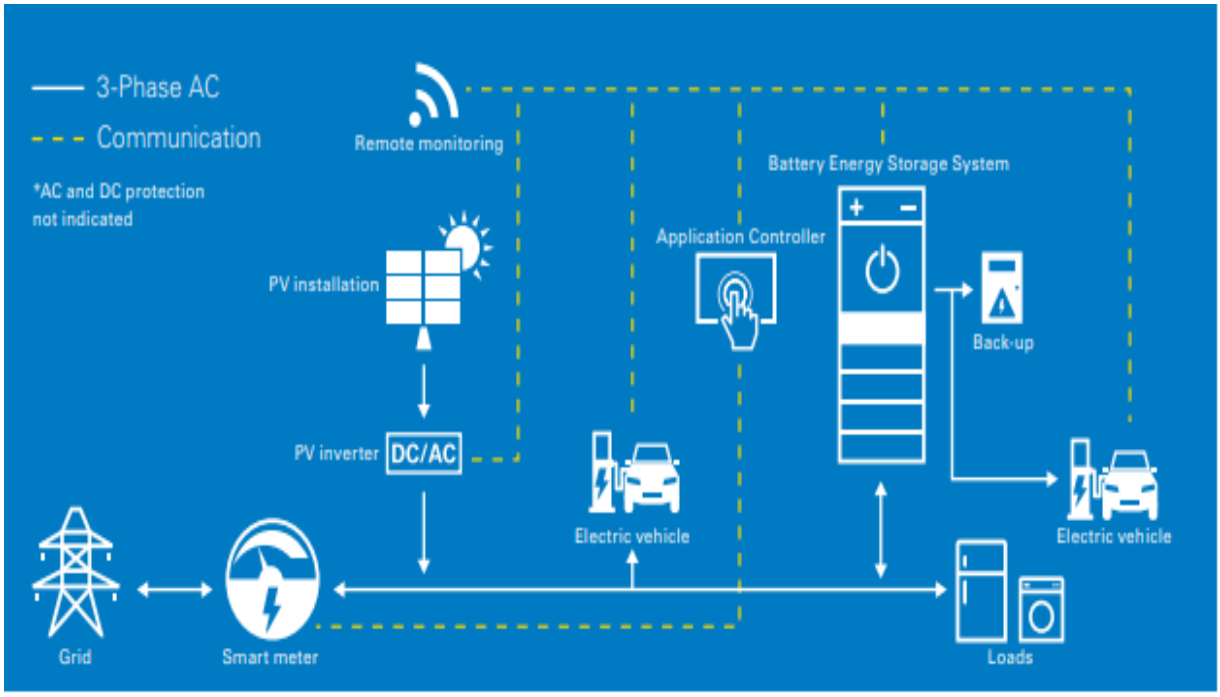

Figure 1. xStorage Compact system functionality in an infrastructure.

xStorage Compact is not so much made for absolute load protection than it is to be used as an energy storage. In most cases xStorage Compact does not even have load and is connected as a parallel system with other UPS devices to not support the whole grid but rather minimise the consumption on the customers

perspective. This means it has different primary functionalities that it must uphold compared to the normal UPS devices.

Typical use case of the xStorage Compact energy storage functionality is measuring buildings total power to the grid to observe if due to photovoltaic installations the power is fed to the grid. In this situation it is better to store this power to the batteries and use the batteries as a power source when the grid power is not produced by photovoltaic installations. This procedure is the principle of photovoltaic self-consumption.

Hardware in the xStorage Compact is based on the 93PS UPS system except for its HMI. Instead, the xStorage Compact uses application controller and system controller to handle user commands that are send through user interface, which allows manual control of the batteries. This eliminates the possibility of batteries being empty in a certain time of day which happens in the UPS, depending how timer has been programmed. See figure 2 below.

Figure 2: xStorage Compact energy storage system.

1.3.1 UPS-as-a-reserve

UPSaaR or Energy Aware UPS technology allows, for example, data centres to also support the electricity grid instead of only consuming power and that results into compensation for the company using Energy Aware UPS devices. With collaboration with Fortum, Eaton has been able to prove UPSaaR feature to work as a part of Frequency Containment Reserve [3], which is an active power reserve to compensate for fluctuations of frequency in the electricity grid. When there is a need for adjusting grid frequency the UPSaaR technology allows the UPS to detect it and discharge the battery back to the grid to regulate the demand.

UPSaaR originates from UPS which was improved and modified for xStorage Compact by adding more features supporting this feature. It is more flexible in its manual operation allowing back and forth power feed to the grid in much larger scale and more often than in the UPS. This is possible due to lithium-ion batteries that are used xStorage Compact which much more suitable for this feature than lead-acid batteries which are used in most of the UPS systems.

1.3.2 Peak shaving

Electricity consuming devices tend to have uneven load profile during day which results to load peaks [4]. By utilizing peak shaving functionalities of the xStorage Compact it is possible to reduce these peaks which result into reduction of power fee. This is possible with using the xStorage Compacts batteries to uphold the demand by discharging during these peak hours and recharging while there is not as much demand for power. See figure 3 below.

Figure 3. How the peak shaving functionality behaves in contrast to power and time.

1.4 Premise

Source code analysed for this thesis is a legacy code made over decades of programming by different programmers. Due to the nature of this project, it is much safer and more efficient to add code for different use cases over the period of the systems life span than to refactor sections of the code to match the set requirements. This results inevitably to running out of memory because the logic is written in a microcontroller that has very limited amount of resources to use.

1.5 Scope

In this thesis the memory issue is analysed, and a solution is presented for safe refactoring of the source code manually. This model presented as a solution can be used as a guideline in any phase of the project life cycle when concern of memory resource limitations come into factor while developing new features or sustaining existing functionalities.

Techniques are implemented based on their viability of the xStorage Compact system and presented based on their usability on embedded systems overall. The techniques in this thesis have their viability and risks assessment presented in a clear manner and the results shown how they affect positively or negatively on the overall outcome of the project. The techniques that have not

been utilized and are presented in this thesis have their usability analysed and their possible use cases explained.

2 Techniques

When making an action plan for this thesis there had to be a decision made on what techniques would be implemented for this project. Most of the optimization methods found are created for performance enhancement in the sense of speed which would be beneficial but for this thesis not viable due to scope. This section does not cover all the techniques available but were chosen based on possible high benefits compared to complexity and viability for this project.

2.1 Used techniques

Techniques described in this section are implemented in the source code throughout of this project and have their effect measured based on compiled code map.

2.1.1 Dead code elimination

Dead code elimination was used in this thesis as a starting point for optimization of the high memory consuming objects. By going through the code and looking for situations where the code is never used or is initialized but never accessed can be cut out which results on reduction of memory usage depending on how often these variables or functions were called.

2.1.2 Common subexpression elimination

Common subexpression elimination is similar technique as using constants to save precomputed results. The idea is to combine common variables and computations in to one which are used multiple times throughout a function or a whole program. [5]

In example [Listing 1] each phase voltage is adjusted with compensation and offset which are common with each of the computations, therefore it is possible to create a temporary variable to do this computation beforehand to reduce the memory overhead caused by each calculation.

Common subexpression elimination can be done by the compiler but in this thesis when this technique is applied a distinct difference in memory consumption is noticed after compiling new code. This means that the compiler is not aware of every possible aspect and situation where the code is implemented in a way that common subexpression elimination could be utilized. [6]

```
void foo( void )
{
    float VoltageA, VoltageB, VoltageC;

    VoltageA = MeasurementA * Compensation * Offset;
    VoltageB = MeasurementB * Compensation * Offset;
    VoltageC = MeasurementC * Compensation * Offset;
}

void foo( void )
{
    float VoltageA, VoltageB, VoltageC;

    //Common subexpression
    float CompOffset = Compensation * Offset;

    VoltageA = MeasurementA * CompOffset;
    VoltageB = MeasurementB * CompOffset;
    VoltageC = MeasurementC * CompOffset;
}
```

Listing 1: Common subexpression elimination example.

2.1.3 Function refactoring

When refactoring code, there is a list of items to consider before and after doing changes to functions to assure that what programmer is doing is safe. [7 p.572]

- Changes need to be done systematically.
- Code should work before the refactoring.

- Keeping refactoring at small portions and one at a time.
- Listing the steps to take doing refactor.
- Retesting.
- Code review.
- Risk assessment.
- Is the change viable performance vice.

2.1.3.1 Ternary operators

Ternary operators are conditional operators that can be used in contrast to if, else statements. It takes 3 operands a condition followed by a question mark, expression to execute if the condition is true and expression if the condition is false. using ternary operators is shown in [Listing 2].

```
void ternary(void)
{

    int a;
    int b;
    int c;
    bool cond = true;

    a = (cond) ? b : c;
}
```

Listing 2: Ternary operator example.

Ternary operators tend to be useful or harmful tool memory vice depending on how they are utilized. In this thesis ternary operators are described in a helpful way of function refactoring. When using ternary operators, it is possible to initialize variable as a constant with two or more possible outcomes. Depending on the situation, the ternary operator can be utilized as a memory saving utility. In this thesis presents an example of useful function refactoring situation using ternary operator, from a badly written conditional [Listing 3], to a disputably well readable, two compact ternary operator-based conditionals [Listing 4]. Ternary operators tend to be described as hard to read and the same benefit of reducing the amount of wasted memory due to additional variables can be obtained without ternary operators as shown in [Listing 5].

```
void Conditional( void )
{
    float tempInputMax;
    if (state)
    {
        if( NominalValue != 0 )
        {
            tempInputMax = NominalValue / 10;
            tempInputMax = Current / tempInputMax;
        }
        else
        {
            tempInputMax = Current / 230;
        }

    }
    else
    {
        float tempNomInput;
        if( NominalValue != 0 )
        {
            tempNomInput = NominalValue / 10;
        }
        else
        {
            tempNomInput = 230;
        }
        tempInputMax = CapCurrent / tempNomV;
    }
}
```

Listing 3: Badly written conditional.

```
void Conditional( void )
{
    const float tempNomInput = ( 0 != NominalValue  ) ?
                               ( NominalValue  / 10.0f ) : 230.0f;

    float tempInputMax = ( state ) ? ( Current / tempNomInput )
                                   : ( CapCurrent / tempNomInput );

}
```

Listing 4: Ternary operator conditional refactoring of previous listing.

```
void Conditional( void )
{
    float tempInputMax;
    float tempNomInput;

    if( NominalValue != 0 )
    {
        tempNomInput = NominalValue / 10;
    }
    else
    {
        tempNomInput = 230;
    }

    if(state)
    {
        tempInputMax = Current / tempNomInput;
    }
    else
    {
        tempInputMax = CapCurrent / tempNomV;
    }
}
```

Listing 5: Previous function refactoring without ternary operators.

2.1.4 Data alignment

Data alignment means a chunk of a data in memory which contrasts with the memory address. If the memory address of the variable that has been addressed to it is a multiple of four so that the address ends with first and second bit being zero, the variable is naturally aligned.

In this thesis a microcontroller with a 32-bit architecture is used which converts into 4bytes which means that if an unsigned integer of 16 bit is followed by a 32-bit integer as a data member, that would cause 16 bits of padding between the 2 variables at compile time.

Data members that have been assigned in a non-logical order so that size of the variables in the order of initialization do not fill up a whole byte results to padding between these members *[Listing 6 & Figure 4]*.

```
struct a
{
    int a1;
    char a2;
    short a3;
}
```

Listing 6: struct with a need for padding.

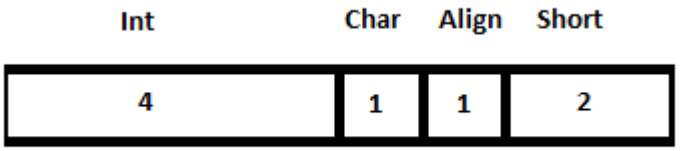

Figure 4: Padded memory structure of listing 6.

2.1.5 Bitfields

The purpose of bitfields is to pack multiple data members inside of a few bytes depending on hardware which can be used only inside of a structured data type. Normal boolean operator variable takes one byte of memory and has two possible options which it stores.

This thesis implemented a refactoring of global bool operations which are used for true or false comparison. The refactoring done takes this principle and converts booleans into a bitfield that is stored into a structure of descriptive name. This method allows us to convert one-byte variables to one-bit variables that store zero or one in them depending on if they need to be false or true, respectively. Structured bitfield takes N multiple of one-byte in every 16 bits assigned where N is a positive integer which means that assigning three Booleans into a bitfield would already result into memory performance improvement.

```
bool a1 = false;
bool a2 = false;
bool a3 = false;
    ...
bool an = false;
```

Listing 7: Initialized booleans.

```
typedef struct
{
    uint16_t a1:1;
    uint16_t a2:1;
    uint16_t a3:1;
    ....
    uint16_t an:1;
}Abool;

Abool A = {false, false, false, ... , false}
```

Listing 8: Initialized bitfield.

2.2 Unused techniques

In this thesis the unused techniques would have cost too much time, decrease readability, have too high of a risk of breaking essential features of the ECM functionalities in contrast to benefits or has already been used throughout the project.

2.2.1 Global constants & lookup tables

Constants are beneficial to utilize predetermined values that do not need be modified after initialization these can be defined with *#define* directive which is a pre-processing macro or by *const* keyword. Globally defining constants in C++ can decrease or increase complexity depending on the use case and help track the memory usage at compile time. Constant variables and definitions that have not been used are ignored by the compiler which results into no memory consumption in their part, for example defining different calculated timer values do not have to be defined based only on use case and do not cause memory overhead if forgotten to be removed.

Using constant variable arrays as tables is an effective way of decreasing complexity and best use of memory in contrast to complex algorithms. Using conditionals to initialize variables instead of constant tables causes more memory overhead and decreases codes readability. Other best use case of is precomputing results and assigning them to constant variables to be accessed

by different functionalities. This decreases memory overhead caused by computations done by the controller and steps taken for acquiring results from different computations. *[7. Chapter 26.4, p.667]*

In this thesis table lookups and precomputing results have not been used for refactoring the code more memory efficient due to source code already utilizing these methods effectively.

2.2.2 Loop unrolling

Loop unrolling is a technique that is used to transform loop to more efficient format [Listing 9] which can be acquired manually or by compiler optimization. This technique does trade-off between speed and space which is why it was not used in this thesis but rather checked in the source code if this technique has been used previously. Overhead of a loop requires a lot of resources which makes opening the loop or running the loop in larger sections much more efficient. [8]

```
void foo( void )
{
    for(int i = 0; i < 20; i++ )
    {
        function(i);
    }
}
// Normal Loop

void foo( void )
{
    for(int i = 0; i < 20; i += 5 )
    {
        function(i);
        function(i+1);
        function(i+2);
        function(i+3);
        function(i+4);
    }
}
// Unrolled loop
```

Listing 9: Loop vs unrolled loop comparison.

2.2.3 Copy propagation

Copy propagation means optimizing code by skipping saving predetermined data to a new variable which would result into predetermined values turning into dead code or adding runtime stack usage by the size of a new variable unnecessarily [Listing 10].

```
int CopyPropagation()
{
    // x assignment can be optimized into returning the value straight
    //int x;
    //x = 1;
    //return x;
    return 1;
}
```

Listing 10: Example of simple copy propagation optimization.

Possible problems occurring from copy propagation exceed the benefit of the using this technique on a legacy code which is why copy propagation was ignored in this thesis implementation. Mostly this method was not used due to it affecting readability of the code because the variables data is assigned to have a describing name to tell the programmers what that specific data is for and because compiler handles this effectively. [9]

The biggest issue with this technique occurs if global data is not assigned in a critical section to a temporary variable as shown in [listing 11]. In this case global variable can be accessed by a different task after it has been checked by a conditional, which could result into undesired results and in worst case scenario dividing by zero which would corrupt the entire program.

```
void CriticalSection()
{
    //Assign global variable to temporary variable
    int tempGlobalVariable = GlobalVariable;

    if(tempGlobalVariable != 0)
    {
        // GlobalVariable might be changed by another task.
        float y = 10 / tempGlobalVariable
    }
}
```

Listing 11: Critical section with a mandatory propagation.

2.2.4 Compiler optimization

The compiler used for xStorage Compact uses TMS320F28335 microcontroller technology which utilizes C2000 real-time MCU control which allows optimization only to a certain extent which can be read from TMS320F28335 optimizing C/C++ compiler user's guide. [10] Due to the lack for consideration of limited code space in embedded system compiler optimization, manual source code size reduction becomes more viable option. Of course, there is a possibility of using a different and more effective compiler but in this situation is not possible due to the scope and type of the project in question because the effort it would take in contrast to benefits.

Optimization of compilers is not enough in most cases due to the difficulty of analysing the amount of code space needed for a fully functional program. Compilers also usually as a last effort provide means of optimizing for size instead of performance but that will require testing of all performance constrained features again if they still match the criteria set and is not a valid option for that reason. [11]

2.2.5 Assembly

Assembly code is a machine code that is the only one that is below C++ code. Converting C++ code to a lower-level code would improve performance significantly and would be a viable option in a small project that does not require constant maintenance.

The xStorage Compact project requires portability between microcontrollers because the project itself is based on 93PS code and the program is required to run on 5 separate UPS systems which some of them have different microcontrollers in their PCS. Utilizing C/C++ code supports portability and Assembly might result into portability problems due to the instruction sets made for one microcontroller having to be modified for each microcontroller separately which is not as efficient as modifying C/C++ code. [12]

3 Dynamic memory allocation

Source code in a large-scale project has in many cases need for compatibility for a lot of similar devices for sustainability reasons. There is a great place to utilize dynamic memory allocation to reserve into memory only the needed HW specific features, variables and constants used for the specific device which it is intended. [6]

3.1 Memory pool utilization

In my innovation project “Dynamic memory allocation in embedded systems 2021” [13] I describe a memory pool as a collection of containers that have a variating fixed size chunks of bits or bytes in them. The memory that has been reserved by the pool can be freely allocated and deallocated to recycle memory at runtime.

In this thesis I propose a way to utilize these memory pools in initialization phase of the code. There are a lot of filter coefficients that are needed to calculate the current and voltage variables and they are utilized as shown in [listing 12] as a constant structure array of float variables. These hold arrays 7*5 bytes of constant float variables which results to taking 70 bytes of memory in the binary code per array. If the coefficients are initialized in a memory pool [listing 13] memory usage can be calculated by a formula (1) where N is actual bytes of the objects and N / 16 is rounded up. [13]

$$RAM = \mathrm{N} + \left(\left\lceil \frac{\mathrm{N}}{16} \right\rceil + 7\right) Bytes \quad (1)$$

In this scenario the memory usage turns out to be 45 bytes of memory that is used by the dynamic memory pool in contrast to 70 bytes. If implemented throughout the code this could accumulate up to 2kB – 4kB of memory that could be saved.

```
const Filter Coefficients [] =
{
    { 0.x,        0.x,      0.x,        0.x,      0.x,        0.x,      0.x   },
//A Module
    { 0.x,        0.x,      0.x,        0.x,      0.x,        0.x,      0.x   },
//B Module
    { 0.x,        0.x,      0.x,        0.x,      0.x,        0.x,      0.x   },
//C Module
    { 0.x,        0.x,      0.x,        0.x,      0.x,        0.x,      0.x   },
//D Module
    { 0.x,        0.x,      0.x,        0.x,      0.x,        0.x,      0.x   },
//E Module
};
```

Listing 12. A C++ example of a structure array holding coefficient values used for different hardware.

```
Filter *Coefficients = NULL;

void Initialize(void)
{
    DynamicPool< Filter, SIZE> Pool;

    Coefficients = Pool.Allocate();

    switch (MyHardwareNumber)
    {
        case A_MODULE:
            *Coefficients = { A COEFFICIENTS };
            break;

             ...

        default:
            *Coefficients = { Default COEFFICIENTS };
            break;

  }
```

Listing 13. A C++ example of initializing coefficient values for different hardware utilizing memory pool.

Issue with this implementation would be the trade-off between flash and RAM memory which is not necessarily optimal because there is less RAM memory available. Because memory pools are statically allocated at compile time there is no issue with finding the microcontroller running out of memory as normally could happen when using dynamic allocation. Memory pools are a working option also because using them is deterministic and does not cause memory leaks.

3.2 Dynamic features

There are lot of features that are included in the source code due to compatibility with other devices that are not used with the xStorage Compact firmware. This means that there are a lot of variables created for settings, measurements and conditions that are meant for these features and go unused while consuming memory throughout the program.

Using dynamic operations on initializing these variables in the constructor depending on the module can reduce RAM memory on each separate hardware significantly with a trade-off to flash memory. Theoretically using a memory pool which size and type is initialized based on the hardware and using those memory locations for feature variables would work. The principle basically works as a memory saving feature due to the program only initializing the memory pool at the compile time and then at run time only saving the variables to the pool that are needed for the system in question.

Risk in this implementation would be overlooking variables that are accessed in wrong hardware which would result into undefined behaviour. Also, in a project that has not planned a memory pool implementation would not be able to freely reserve memory from the planned scope to adjust to the requirements placed by the memory pool or at least it would require major refactoring.

4 Assembler

TMS320C28x C/C++ compiler accepts ANSI standard C/C++ source code and produces assembly language source code for the TMS320C28x based devices. [14] Data is divided in a variety of memory segments as seen in figure 5. Segment .data usually contains the initialized data, this is in EABI, which is what is used in the target system of this thesis. COFF has its initialized data section called .cinit. Uninitialized data is saved in .bss and .ebss section for EABI and COFF respectively. [14]

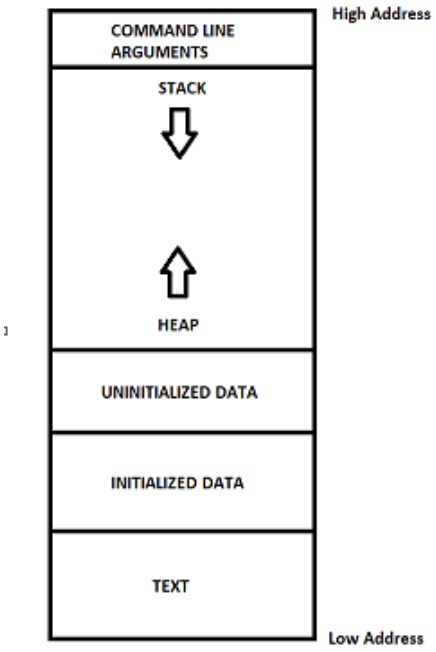

Figure 5: Memory segmentation in C/C++.

5 Implementation & Results

This section of thesis goes through the plan of action that was used throughout the project and the results that were recorded each build cycle after implemented change to source code.

5.1 Tools

Source code for ECM is compiled in code composer studio version 5.1 which is eclipse-based compiler and uses C2000 real-time 32-bit microcontroller. Real-time functionality is achieved by Texas instruments DSP/BIOS real-time operating system. Testing of the compiled code was done in a xStorage Compact 40-kilowatt unit at test laboratory of Eaton company building. Version control and code changes were done step by step by committing to bitbucket to track the progress and for easy maintaining of the source code.

For checking preliminary viability of different techniques and comparing methods on a real environment easily without building large project a

LPCXpresso 1549 was used. Texas instruments LPCXpresso projects were compiled with MCUXpresso IDE v10.2.1 which is also an eclipse-based IDE. The testing project used LPCOpen middleware library and freeRTOS operating system.

5.2 Implementation

Plan of action for this thesis was to research for beneficial and simple to utilize coding techniques used for C++ to reduce overall memory consumption of the ECM source code. Starting point was to test the potential effects of the techniques in the source code and testing project created on the LPCXpresso 1549 microcontroller. This was done so that the irrelevant techniques could be excluded in the next phase of implementation so that only the beneficial methods were utilized.

Pareto principle which states that roughly 80 percent of issues come from 20 percent of the causes is accurate description that shows in source code. Order of refactoring the code in each optimized class was done based on this principle. By looking at the few of the most memory consuming objects that could be observed from memory map created by compiler. These most consuming objects would be handled one at a time based on observations. See figure 6 below.

Techniques in the work phase of this thesis was not done in any order apart from executing dead code elimination for each of the objects first so that refactoring later would not accidentally be done for unused code. Parts of testing the techniques that were done on LPCXpresso microcontroller showed which of the techniques had most potential of working for this project combined with code analysis.

If the refactoring done had bugs or did not decrease the memory usage it was either fixed or reverted to the original implementation. Results of this thesis

does not consider reverted changes due to them not affecting the outcome of the work phase.

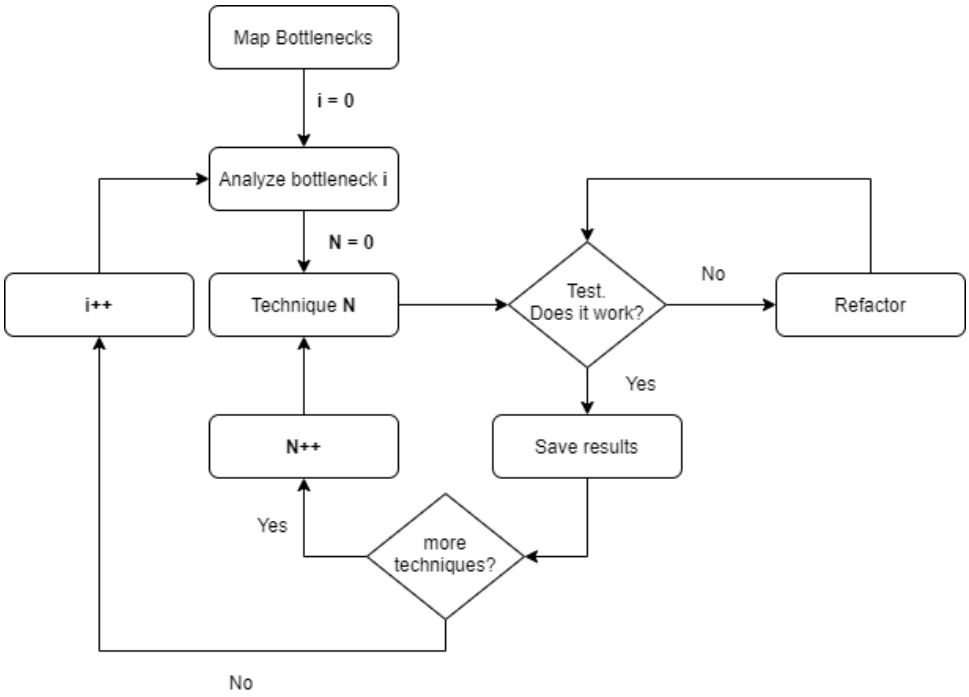

Figure 6: Work flowchart.

5.3 Results

Results show that the dead code elimination was the most effective way of optimizing the program and highest memory consuming objects were prone to be optimized the most which was hypothetically accurate at the start based on pareto principle. Flash memory consumption was reduced more than RAM memory which was expected due to flash being larger and used more.

Memory consumption for RAM in the beginning of the implementation was 94,577% of the overall available RAM memory and the flash consumption at the start was 61,242% of the overall available flash memory. At the end of the technical solution RAM and Flash memory consumptions were 94,5769% and 60,759% which results into approximately 1,5% and 0,5% improvements respectively. The result itself is decent and the memory that has been saved can be used to implement a few relatively small features which can be deducted

from experience. If techniques are utilized throughout the entire program the result could be approximately twice as large.

Techniques implemented were focused mostly on the highest consuming classes in the source code due to scheduling limitations which is why only few objects are listed in the result diagrams *[Figure 7-10].* Target classes were picked based on the amount of RAM memory they consume due to RAM memory being more expensive than flash. Flash diagram of class-based memory reduction in figure 7 shows how the most consuming three objects regarding inverter, rectifier and battery state have the most impact on the overall result. Figure 8 shows how the much flash memory was saved in bytes based on technique used and figures 9 and 10 show RAM memory reduction in the outcome based on class object and technique respectively.

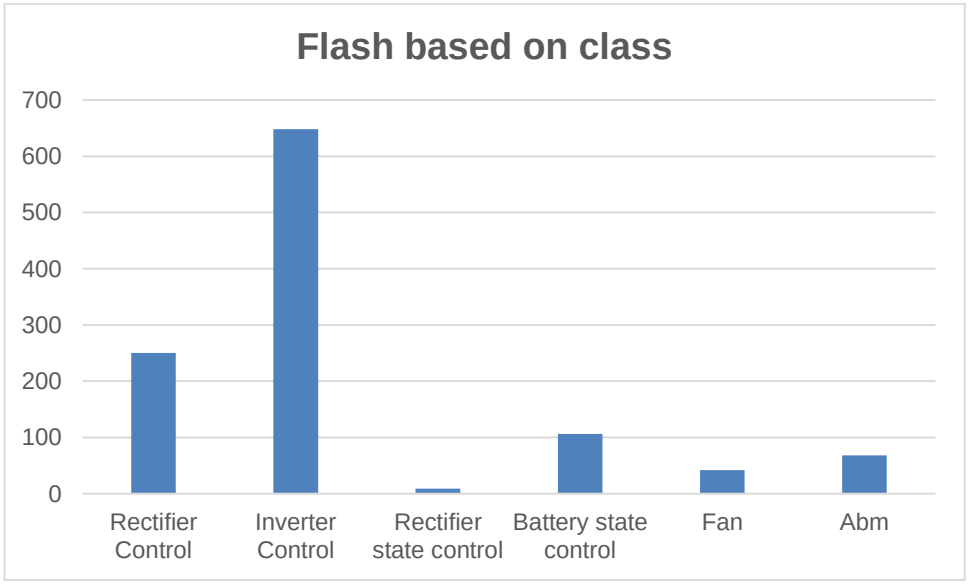

Figure 7: Flash memory reduction in bytes based on object.

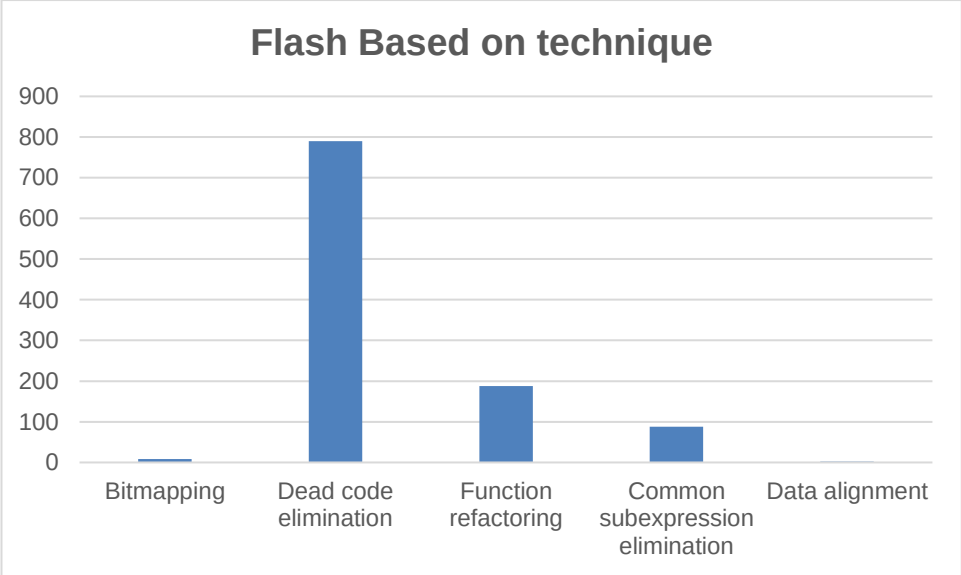

Figure 8: Flash memory reduction in bytes based on technique.

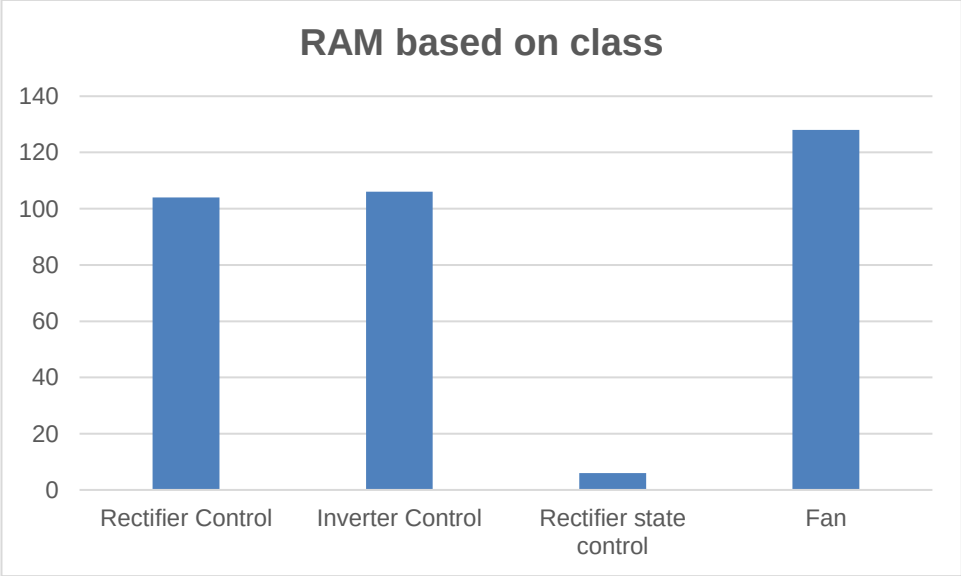

Figure 9: RAM memory reduction in bytes based on object.

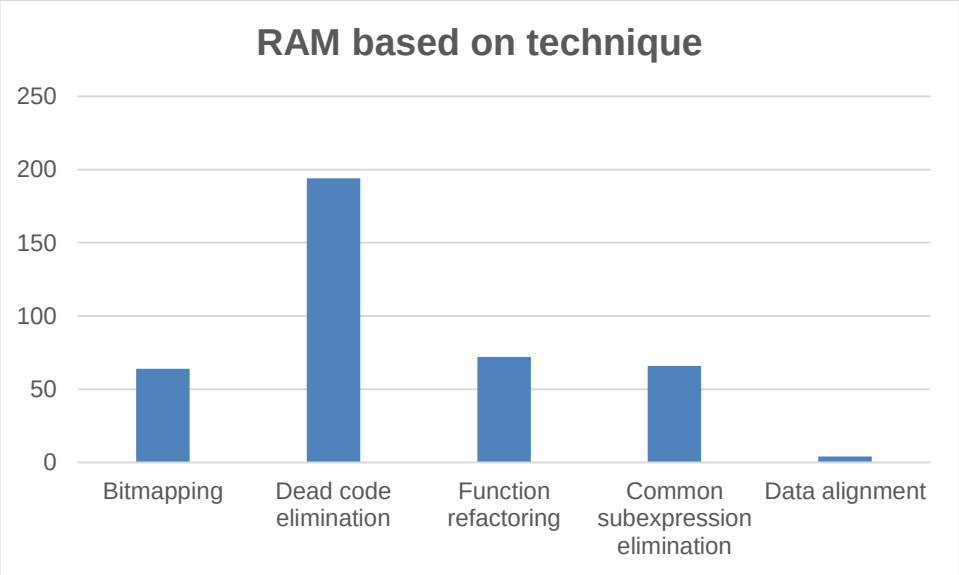

Figure 10: RAM memory reduction in bytes based on technique.

6 Conclusion

This thesis provides a working model for refactoring source code in a large-scale project that can be utilized based on the results as a programmer sees fit. These techniques have the potential to be used effectively when the preliminary research and testing has been conducted based on this thesis which provides a faster way of memory-based optimization in the target code compared to self-analysis.

In xStorage Compact which is in the sustaining phase of the project it is not practical to start adding external memory to the project or changing the microcontroller for more memory. If this were done it would mean that all the customer devices require to have their hardware changed on the field which would require to shutdown the systems for the duration of maintenance or rebuild and then replace the field systems which is much less optimal than doing refactoring to make room for a new sustaining firmware that is just updated in the field.

This improvement presented in this thesis could uphold two decent sized firmware feature updates and when considering the time, implementing the

following techniques described in this project it would approximately take 2 days of work to implement. From experience it can be stated as an example that a recent decent size firmware update that implements parallel system load share calibration took only half of the amount of memory that has been saved in the practical portion of this thesis.

Techniques presented in this thesis can be expanded into the other systems and projects that are made and still sustained at Eaton. That covers most of the most profitable systems that are currently on the field and sold to customers and which have their firmware modified constantly.

For improvement dynamic memory allocation possibilities should be examined more throughout due to the vast possibilities it has. It is very time consuming with high-risk high-reward results and cannot be utilized in immediate use which is why this procedure was not investigated more in this thesis.

Prioritising refactoring of functions that are moved to the ramfuncs section in the code would have been a beneficial addition in the workflow of the implementation. When refactoring and improving these functions, the source code is guaranteed to have higher impact on the RAM consumption.

Bitfield usage has high potential to be utilized in the source code and could have been used more throughout the project which might have resulted in better overall improvement. However, it was not done during the work phase of this thesis due to the disadvantage of bitfields not supporting pointer assignment which would require more throughout analysis of the overall requirements of each refactored variable.

7 References

[1]. A. R. Mahajan & M. S. Ali. Optimization of memory system in Real-Time Embedded Systems. July 28th, 2007. Online paper. WSEAS. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.416.6803&rep=rep1&type=pdf. Read April 21st, 2021.

[2]. Eaton corporation home page, web material. Eaton. https://www.eaton.com/gb/en-gb/company. Read June 12th, 2021.

[3]. Eaton corporation website. EnergyAware UPS – Green Money in Grey Spaces. Web material. Eaton. https://powerquality.eaton.com/EMEA/UPSaaR/default.asp. Read May 8th, 2021.

[4]. Karmiris Georgios & Tenger Tomas. Peak shaving control method for energy storage. ABB AB, Corporate research center. Online paper. https://www.sandia.gov/ess-ssl/EESAT/2013_papers/Peak_Shaving_Control_Method_for_Energy_Storage.pdf. Read May 8th, 2021.

[5]. Platzer André. Lecture notes on basic optimizations. Online lecture notes. Chapter 4. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.470.1843&rep=rep1&type=pdf. Read August 7th, 2021.

[6]. Hosangadi Anup, Fallah Farzan, Kastner Ryan. Factoring and eliminating common subexpressions in polynomial expressions. 2004. Online paper. Chapter 1. http://cseweb.ucsd.edu/~kastner/papers/iccad04-subexpr_polynomials.pdf. Read August 7th, 2021.

[7]. McConnell C. Steven. Code complete second edition. 2004. Book. Microsoft corporation. Read April 20th, 2021.

[8]. Ekström Viktor. Manual micro-optimizations in C++. Malmö University. 2019. Bachelor thesis. Chapter 1.1.2, p.10. https://www.diva-portal.org/smash/get/diva2:1480520/FULLTEXT01.pdf. Read August 7th, 2021.

[9]. Fog Agner. Optimizing software in C++. Technical university of Denmark. 2021. Online paper. P.68. https://www.agner.org/optimize/optimizing_cpp.pdf. Read August 7th, 2021.

[10]. TMS320C28x Optimizing C/C++ Compiler v21.6.0.LTS. Texas instruments. 2021. User's guide. Chapter 3.2, p.55. https://www.ti.com/lit/ug/spru514w/spru514w.pdf?ts=1628318958477. Read August 7th, 2021.

[11]. Abdulla Fadle Mohammed. An efficient manual optimization for C codes. ICSRS Publication. Online paper. 2010. http://emis.impa.br/EMIS/journals/IJOPCM/Vol/10/IJOPCM(vol.3.2.10.J.10).pdf. Read August 7th, 2021.

[12]. First steps with embedded systems. Byte craft limited. Online book. 2000. Chapter 1.1. https://www.phaedsys.com/principals/bytecraft/bytecraftdata/bcfirststeps.pdf. Read August 7th, 2021.

[13]. Vainio Ville, Dynamic memory allocation in embedded systems. 2021. Innovation project. Eaton. Read April 27th, 2021.

[14]. TMS320C28x Assembly Language Tools v18.12.0.LTS. 2001, revised 2019. User's guide. Texas instrumentals. p.19. https://www.ti.com/lit/ug/spru513r/spru513r.pdf?ts=1599377715393. Read May 29th, 2021.

7.1 Photo references

Figure 1&2: Eaton corporation website. xStorage Compact Single rack energy storage system. 2020. Web leaflet. Eaton. https://www.eaton.com/content/dam/eaton/products/energy-storage/xstorage-compact/en-us/eaton-xstorage-compact-leaflet.pdf. Read May 8th, 2021.

Figure 3: Karmiris Georgios & Tenger Tomas. Peak shaving control method for energy storage. ABB AB, Corporate research center. Online paper. https://www.sandia.gov/ess-ssl/EESAT/2013_papers/Peak_Shaving_Control_Method_for_Energy_Storage.pdf. Read May 8th, 2021.