Exploring the fundamental differences between compiler optimisations for energy and for performance

James Pallister

Supervisors: Kerstin Eder
Simon J. Hollis

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

March 2016

43,000 words.

This page is intentionally blank.

Abstract

The efficiency of an application can have a huge effect on how long a device will run on its batteries. With no significant increases in battery capacity, yet applications requiring longer battery life, it falls to software to be more efficient with its use of energy. This is particularly salient in deeply embedded systems which may have to run on batteries for years.

The high-level nature of writing a computer program detaches the programmer from the underlying hardware, allowing programmers to quickly write code that will be functional on a diverse set of devices. However, it also complicates writing efficient software, since the code may go through many transformations before it is executed, each of which can result in larger energy consumption without this being easily observable. Therefore, methods of increasing software energy efficiency are needed.

Compiler optimisations provide an ideal way to achieve this, providing the ability to automatically transform software into a more efficient form. Typically, compilers have focused on making an application fast — there exist hundreds of optimisations to decrease runtime, and often these are effective at reducing energy. However, few optimisations exist to specifically decrease energy consumption.

This thesis explores the differences between automated ways to reduce energy consumption and execution time at the compiler level. By exploring an extensive selection of existing compiler optimisations, through statistical techniques and genetic algorithms, it is discovered that current optimisations only reduce energy consumption because they reduce the execution time; while the optimisations affect the power dissipation of the device, this effect is incidental.

The lack of optimisations which affect power dissipation implies a class of compiler optimisations has been missed from conventional compilers. This thesis develops two new optimisations belonging to this class. To create these optimisations, low-level hardware-specific energy characteristics are identified, focusing on embedded systems. The characteristics are rigorously modelled, allowing the compiler to make optimisation decisions at a higher level of abstraction. One of the optimisations manages to save up to 26% of the energy consumption, achieving this by focusing on average power reduction, rather than execution time reduction.

The combination of the optimisation for energy with the existing optimisations for time is explored and found not to interact significantly with other optimisations, proving to be linearly composable. This again suggests that the energy optimisation belongs to a different class of compiler optimisation.

To achieve further energy efficiency gains, a thorough vertical integration process needs to be introduced. This requires identifying (possibly) hardware-specific energy characteristics, modelling them at a level accessible to the compiler, and then making optimisation decisions based on these models. This level of integration is essential to bridge the levels of abstraction that exist between the hardware and software, and allows compilers to be successful at reducing applications' energy consumption and consequently increasing battery life.

This page is intentionally blank.

Acknowledgements

My PhD has been a great experience, allowing me to explore interesting topics and delve deeply into areas which I would otherwise never get chance to experience. First, I would like to thank my supervisors, Kerstin Eder and Simon Hollis for their continued direction, support and encouragement with every aspect of my PhD. I'd like to thank my external examiners, Alan Mycroft and Michael O'Boyle, for the effort they put into reviewing my thesis, and thoughtful feedback during the viva.

I would like to thank all my friends and family for their support — in particular Steve Kerrison, Jeremy Morse, Dan Curran, Jake Longo, Craig Blackmore, Kyriakos Georgiou, Tom Deakin and Dejanira Araiza Illan. In particular I'd like to thank Joanna Hopping, for much emotional support during the course of my PhD.

During the summers of my PhD I was hosted by Embecosm. I am very grateful to Jeremy Bennett, Simon Cook and everyone at Embecosm for hosting me and working with me on interesting projects — it was a pleasure to work with everyone there.

Finally, I am grateful to EPSRC for providing me with funding, without which I would not have been able to complete this PhD.

This page is intentionally blank.

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

Signed: _____

Date: _____

This page is intentionally blank.

Contents

List of Tables

List of Figures

List of Publications

James Pallister, Simon J. Hollis and Jeremy Bennett. "Identifying compiler options to minimize energy consumption for embedded platforms". In: *The Computer Journal* 58.1 (Nov. 2013), pp. 95–109.

James Pallister, Kerstin Eder, Simon J. Hollis and Jeremy Bennett. "A high-level model of embedded flash energy consumption". In: *CASES'14 Proceedings of the 2014 international conference on Compilers, Architecture, and Synthesis for Embedded Systems.* New Delhi, India. ACM Press, 2014, p. 74.

James Pallister, Kerstin Eder and Simon J. Hollis. "Optimizing the flash-RAM energy trade-off in deeply embedded systems". In: *CGO'15 Proceedings of the 2015 international symposium on Code Generation and Optimization.* San Francisco, USA. ACM Press, 2015.

James Pallister, Simon J. Hollis and Jeremy Bennett. "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms". In: *arXiv, CoRR.* 2013.

This page is intentionally blank.

Chapter 1. Introduction

Energy consumption is one of the most critical metrics for embedded systems, given that battery life has not improved at the same rate as the power requirements for our devices. Software is in a prime position to reduce this energy consumption by enabling smarter ways to control the processor and its peripherals. Optimisations to reduce a program's runtime or energy consumption are often complex, but can be done automatically by compilers and tools integrated with the compiler, reducing the effort required from the developer. For example, previous attempts have used the compiler's existing optimisations to reduce energy consumption [1], as well as attempting to create new optimisations which reduce energy by instruction scheduling [2], instruction selection [3], inserting sleep modes [4] and scratchpad memory utilisation [5]. All of these optimisations must balance energy consumption and execution time, while keeping the code size small enough to fit in the system's memory.

This thesis targets embedded systems, ranging from the deeply embedded processors found in devices designed to run on batteries for years, to higher-end embedded devices with moderate processing requirements. Battery life, and thus energy consumption, is particularly important to these devices. These devices are also heavily constrained by the available memory to store code and data, since the smallest of the embedded devices tested has just 16kB of flash and 2kB of RAM. These memories are accessed directly, without caches.

All of the embedded systems have a processor at the heart of the SoC, controlling and coordinating all of the available functionality (typically deeply embedded systems are not multi-threaded). The SoCs and their processors have a range of different features, such as differing memory technologies, pipeline structures and architectures. The large variety of features in these deeply embedded systems means that code and even optimisations which apply to one system may be ineffectual on another, and comparing two diverse systems is challenging. To partially mitigate this problem, this thesis creates a benchmark suite with characteristic applications to run on all the platforms. The suite is designed to have desirable characteristics for evaluating compiler optimisations and energy on deeply embedded systems.

The actions of an application can have a large effect on the overall performance and energy consumption. By optimising the program, a desirable trade-off between the various metrics can be made. Typically, these metrics include the execution time of a task, its energy consumption and its code size. Occasionally other metrics are important, such as Worst Case Execution Time (WCET) [6] and peak power dissipation. The trade-off between these metrics is complex, and when modifying code to optimise a single metric, all other metrics may change in positive or negative ways. This is exemplified with energy consumption (E), which is the product of what exactly the code is doing (power dissipation, P), and how long it takes to do it (execution time, T),

$$E = P \times T. \tag{1.1}$$

Therefore, simply making the code faster (decreasing T) does not necessarily lower the energy consumption, if the speed-up is caused by replacing energy-efficient operations with energy intensive ones (increasing P).

In general, this implies that optimisations targeting energy consumption need to be more intelligent than optimisations for performance. Since energy consumption is proportional to both execution time and power dissipation, an optimisation must be careful to consider the transformation it makes, as well as how long the resultant code take to execute. This has led to

many previous optimisations for energy actually delivered by improvements in run time [7]. For example, scratchpad memories can be used in place of caches, due to their hardware simplicity and speed compared to direct access to main memory. Much of the benefit in using scratchpad memories comes from their speed, rather than a significantly lower energy consumption. Similar effects are seen in many other studies which reduce memory accesses[1] [8, 9] or utilise SIMD instructions [10] — a large proportion of the improvement is from a reduced run-time.

This thesis explores the difference between optimisations for performance and optimisations for energy consumption, hypothesising that **optimisations specifically designed to reduce energy consumption are significantly more effective at reducing energy than attempting to utilise existing optimisations.** To study this hypothesis, all the existing optimisations implemented in a popular compiler (GCC) are explored, along with combinations of those optimisations. As expected, many optimisations are found to improve performance (i.e. lower execution time), and this improvement leads to a reduction in energy consumption. However, this improvement is from a reduction in total work, rather than achieving a lower average power during program execution.

As a new contribution in this area, two new optimisations are proposed, each having differing trade-offs between energy consumption and execution time. They use features of the SoC not typically considered for performance optimisation. The optimisations are evaluated in detail and found to save significant amounts of energy (up to 26%), suggesting that optimisations specifically targeted towards energy consumption *can* have a significant impact. These optimisations specifically for energy consumption are often system-dependent, needing to exploit a facet of the System on Chip (SoC) that lowers the average power, such as choosing a low power memory or aligning code in such a way that minimises energy consumption. This results in lower energy that is due a reduction in power used by the device, rather than decreasing execution time.

1.1. Context

Many previous studies have attempted to explore existing optimisation effects on energy consumption, and develop completely new optimisations targeting the reduction of energy. This thesis overlaps the areas of compiler optimisations and energy efficiency, taking ideas from the construction of energy models and using them to optimise software for embedded systems. Other ideas are taken from studies exploring the combination and composition of compiler optimisations, in ordering and selecting transformations.

This contribution fits into a broad spectrum of work considering energy modelling. The technique examines how each instruction executed in the processor of an embedded system can be modelled and uses this model to predict the energy consumption of code, without needing to instrument and measure the code's energy consumption on a physical platform [11, 12, 13, 14]. Typically, these models are applied to simulation traces, however static analysis [15] can be used to estimate the energy consumption and the worst case energy consumption [16]. These models are the foundation for many further studies which attempt to minimise a system's energy through modifications to the operating system task scheduling [17, 18], memory prefetching [19] and routing in wireless sensor networks [20].

While these energy models are useful for estimating the total energy consumption of a program, they are not always suited for use in optimisation. An instruction-based modelling approach fundamentally assumes that the choice of instructions can be adjusted to consume a lower amount of energy. In many processors there are other significant elements that affect

[1] Memory accesses in general are higher power than an average arithmetic instruction, however this relative difference is often small compared to how much longer a memory operation can take.

energy, such as peripherals, RAM, and flash, as well as the interconnect joining everything together; these all have larger impacts on the power than selecting different instructions. This has led to higher-level models being developed to take into account some of these other factors, such as the memory hierarchy [21], caches [22] and power states of each block in an SoC [23].

Many studies consider energy consumption at the very lowest level — the number of bit-flips required from each transistor to perform an operation. This has lead to various studies attempting to minimise the number of bit-flips, by instruction scheduling [2], strength reduction [24], operand ordering [25] and changing the encoding of the instruction set [26]. The savings achieved by these studies are often small, and in some cases can negatively impact the execution time, negating the energy savings.

Although this thesis focuses on how modifications can be made to the software, modifications to the hardware that are then properly supported by the software can be very effective. Techniques such as pipeline gating to control speculative execution [27], placing cache-lines in a low energy mode [28], and partitioning the register file into hot and cold (low power) regions [29] all modify the underlying processor, and require compiler support to achieve their energy efficiency. Hardware modification can be assisted with software modelling: various tools developed to explore the energy consumption requirements of a processor before it has been built, such as Wattch [30] for processor design exploration, and CACTI [31] for caches.

While existing studies have looked at energy modelling and compiler optimisations independently there has been little work which combines them, evaluating how effective compiler optimisations for energy can be, and their relationship to performance optimisations.

1.2. Contributions

This thesis poses several research questions, categorised into how existing compiler optimisations affect energy and time, the development of new optimisations which specifically target energy consumption, and the efficacy of energy optimisations when combined with the existing compiler optimisations targeting runtime or code size. These research questions are listed in Chapter 2, and the answers to them form the following major contributions of the thesis. First-author publications supporting the work are also given below.

Development of a benchmark suite to evaluate energy consumption [32]. In Chapter 3, a benchmark suite is developed and evaluated for its suitability for testing energy consumption on embedded platforms. The benchmark suite, BEEBS, is described in a paper and consists of 10 benchmarks from different application categories. The benchmarks are analysed based on their instruction distribution across several different SoCs, ensuring that a good distribution is realised even when compiled for differing architectures. The work describing the benchmark suite is also available in,

> [32] James Pallister, Simon J. Hollis and Jeremy Bennett. "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms". In: *arXiv, CoRR*. 2013.

Analysis of existing compiler optimisations' effects on energy [33]. Chapter 4 discusses the existing optimisations in a compiler and their effect on the energy consumption of several platforms. The analysis uses a technique called fractional factorial design to explore the large number of combinations and find cases where the energy consumption or execution time is influenced by the optimisation. In the majority of cases reducing execution time also reduces energy consumption, however, there are cases where this does not hold true. The energy consumption patterns are related to the architecture of the processor targeted, and it

is found there is no single set of optimisations which is effective for all SoCs or benchmarks. The work is published in,

> [33] James Pallister, Simon J. Hollis and Jeremy Bennett. "Identifying compiler options to minimize energy consumption for embedded platforms". In: *The Computer Journal* 58.1 (Nov. 2013), pp. 95–109.

Development of new compiler optimisations targeting energy [34, 35]. Two new optimisations are developed, targeting energy consumption. The first optimisation builds a model of how energy consumption changes at different code alignments in flash, then uses this model to optimise the code. The modelling work is published in,

> [34] James Pallister, Kerstin Eder, Simon J. Hollis and Jeremy Bennett. "A high-level model of embedded flash energy consumption". In: *CASES14 Proceedings of the 2014 international conference on Compilers, Architecture, and Synthesis for Embedded Systems.* New Delhi, India. ACM Press, 2014, p. 74.

The second optimisation exploits a characteristic of some deeply embedded processors, where executing code from RAM takes less energy than executing code from flash. The optimisation moves carefully chosen basic blocks into RAM, while considering the necessary trade-offs between code size, execution time and energy consumption. The work is published in,

> [35] James Pallister, Kerstin Eder and Simon J. Hollis. "Optimizing the flash-RAM energy trade-off in deeply embedded systems". In: *CGO'15 Proceedings of the 2015 international symposium on Code Generation and Optimization.* San Francisco, USA. ACM Press, 2015.

Analysis of combining optimisations for energy with optimisations for performance. The combination of a particular energy optimisation and the existing optimisations is extensively explored, finding that the optimisation composes linearly in almost all cases, indicating there are few interactions between the energy optimisation and the existing optimisations for execution time. The combinations are evaluated using fractional factorial design, as well as applying it on top of optimisation sets discovered with a genetic algorithm.

In addition to the academic contributions, the benchmark suite, BEEBS, has started to see industry take-up, being used in system evaluation by Xilinx [36], research projects [37], and other academic publications [38, 39]. The benchmark suite is also integral to the MAchine Guided Energy Efficient Compilation (MAGEEC) project, which uses the benchmarks as training data for a machine learning compiler [40].

To enable the gathering of energy results for this thesis, an accurate measurement system was implemented. This custom solution utilised an 'energy-shield' mounted on-top of a host processor board. The energy-shield contains shunt resistors and amplification circuitry to provide the host board with analog signals representing the current and voltage, which can then be digitised and returned to the computer via USB. The firmware and software for these boards was developed by the author, and several events held, promoting the use of the board and educating developers on how to take energy measurements.

Both the measurement board and the benchmark suite have been used in the following research projects involving the author:

MAchine Guided Energy Efficient Compilation (MAGEEC). The MAGEEC project extends GCC [41] and LLVM [42] to choose compiler optimisations based on features extracted

from the application being compiled. Since the efficacy of an optimisation is heavily dependent on the structure of the code to which it is being applied, whether or not to apply an optimisation is decided via machine learning. The machine learning made use of BEEBS as training programs, extracting features and measuring the energy consumption of the benchmarks.

ENergy TRAnsparency (ENTRA). The ENTRA project seeks to provide the programmer with advance knowledge of a program's energy consumption, without the requirement of running and measuring the program. This is achieved by using a compiler-based static analysis, combined with energy models [43, 44].

Finally, a compiler optimisation for energy consumption has been implemented by Embecosm in the GCC compiler, using the work in this thesis (the optimisation and results are covered in Section 5.3.2).

1.3. Thesis structure

The remainder of this thesis is formed of six chapters. Each chapter is self contained, with the relevant literature review appearing at the beginning of each chapter, and the rest of each chapter contains analysis and further develops the topic.

2. **The effect of compiler optimisations on energy and time.** The relationship between time and energy is a key topic of the thesis, and introduced in this chapter. The chapter discusses how an optimisation's effect on power and time affects the overall energy of a program, and poses research questions that the remainder of the thesis answers. These questions lead to the exploration of the difference between optimisations for execution time and optimisations for energy consumption.

3. **Benchmarking.** The benchmarking chapter first gives background on how the energy, time and other metrics are defined when comparing optimisations for energy and optimisations for time. Then, the set of SoCs/platforms used throughout this thesis is presented and the justification and creation of a benchmark suite is described. The benchmark suite, BEEBS, is used throughout this thesis to evaluate execution time and energy consumption for each of the platforms. The chapter discusses why a new benchmark suite designed to measure energy is needed and analyses each of the benchmarks.

4. **Optimisations designed for execution time.** This chapter extensively explores existing optimisations designed for computational throughput and their effect on both energy efficiency and execution time. Initially, the overall optimisation levels available in the compiler are explored. Then, a large exploration of all individual optimisations and their interactions is performed, showing the efficacy of each optimisation, and the most effective sets of optimisations for each combination of benchmark and platform.

   An approximation of the best possible energy consumption and execution time is found using a genetic algorithm and compared to the sets of the best optimisations found.

5. **Optimisations designed for energy consumption.** This chapter discusses the design, implementation and evaluation of two novel optimisations for energy. A model of the memory access pattern's effect on flash memory consumption is developed for the first optimisation. A set of three similar optimisations is proposed to exploit this model.

   A second optimisation, the RAM overlay optimisation, exploits the different in energy between RAM and flash by moving code between the two memories. First, a whole

program model is developed to approximate the energy cost. Second, the implementation of the optimisation is discussed. Finally, the optimisation is evaluated on the benchmark suite (BEEBS).

6. Combining optimisations. This chapter analyses the combination of optimisations for performance and optimisations for energy consumption, building upon the analysis in the preceding chapters. The RAM overlay optimisation is extensively evaluated in combination with the optimisations that already exist in the compiler, answering questions about the composability of energy optimisations with time optimisations and the overall impact on both the energy and time of an application.

7. Conclusion. This chapter presents the conclusions to the thesis, summarising each chapter and the answers to the research questions posed in Chapter 2.

Appendices

The appendices contain additional information on the background of the optimisations discussed, and information about the results obtained.

A. Optimisation reference. This appendix describes many of the optimisations referenced throughout the thesis, giving working examples of the transformation, as well as the intended benefit of the optimisation.

B. Datasets. The best sets of optimisations found by the genetic algorithms are listed here for each benchmark, along with each goal function.

Chapter 2.
The effect of compiler optimisations on energy and time

Each computation that is run on an embedded System on Chip (SoC) will take a certain amount of energy and time. When optimising that computation with a compiler, each optimisation will affect both the energy consumption and the execution time. For optimisations that already exist in modern compilers, the majority target an increase in computational throughput — either reducing the number of instructions necessary to perform the computation, or restructuring the code so that it can execute faster. Other optimisations target code size.

These optimisations do not inherently target energy consumption — any change in energy consumption is purely a side effect, due to energy's dependence on time. The equation below specifies the fundamental relationship between the energy consumption, average power dissipation and execution time of an application, a,

$$E_a = P_a \times T_a, \tag{2.1}$$

where energy (joules) of the program, E_a, is the product of power (watts), P_a, and time (seconds), T_a. This thesis develops the following formulation of an optimisation's effect — when an optimisation is applied to the program, the time, the power and the resultant energy, E'_a, are changed,

$$T'_a = k_T \cdot T_a \tag{2.2a}$$
$$P'_a = k_P \cdot P_a \tag{2.2b}$$
$$E'_a = T'_a \times P'_a = (k_T \cdot T_a) \times (k_P \cdot P_a), \tag{2.2c}$$

where the scaling factors k_P and k_T represent how much the optimisation changes power and time, respectively. Currently, the majority of existing optimisations attempt to minimise k_T — any effect on k_P appears to be by chance. An optimisation for energy will attempt to minimise k_P, possibly resulting in a lower energy consumption without an increase or decrease in execution time. Overall, this means energy consumption is reduced if the inequality holds,

$$(k_T \cdot k_P) < 1. \tag{2.3}$$

The constraint is necessary, since a lower power can be achieved by applying an optimisation that purely extends the execution time, such as reordering the instructions to cause an increase in the number of stall cycles. While this does satisfy the objective of lower power, it likely increases both execution time by a larger amount than it reduces the power dissipation, thus increasing overall energy.

This thesis hypotheses that the existing optimisations in a production level compiler are successful at reducing energy consumption by reducing k_T, and have a minimal effect on k_P. A further hypothesis is that there is a class of optimisations which enable compilers to reduce energy by reducing k_P.

The majority of current research suggests that compilers are only effective at reducing energy consumption by also decreasing execution time [7], indeed many studies which advertise a saving of energy primarily achieve the reduction by decreasing the execution time of the application. This contradicts the other often-held belief that compiler-directed dynamic frequency and voltage scaling can reduce energy consumption by slowing the program's execution and

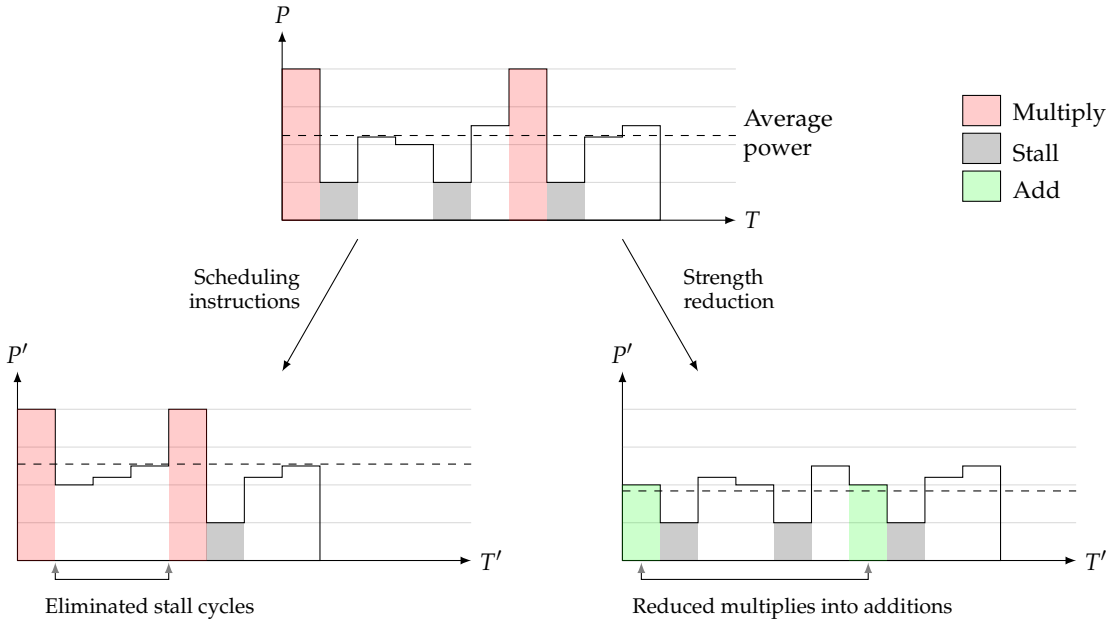


Figure 2.1: Power traces as instructions execute, before and after application of two different optimisations.

reducing the voltage [45]. Analysing current and potential future optimisations' effects on k_T and k_P allows these beliefs to be tested and resolved.

Figure 2.1 shows an example of how the k_T and k_P coefficients change when applying different optimisations. Each of the three diagrams show a power trace over time, at the instruction level granularity. The instruction stream consists of high-power multiplies, low-power stalls and other instructions of average power. The instruction scheduling optimisation (see Appendix A, page 102) removes stall cycles by reordering instructions, decreasing the overall execution time — it has a large effect on k_T. However, removing the stall cycles also raises the average power, leading to only a moderate effect on energy. Another optimisation, strength reduction (see page 106), replaces the high power multiplies with lower power additions, having the effect of reducing average power, k_P, while keeping k_T constant. Overall, the strength reduction achieves a lower energy consumption in this (contrived) case.

2.1. Research questions

The main research questions answered in this thesis are given below. The questions are roughly categorised by the chapter in which they are answered.

Standard compiler optimisations.

- Do existing compiler optimisations save energy purely by reducing the k_T coefficient?
- Are there instances of standard compiler optimisations which affect k_P?

Compiler optimisations for energy.

- Is there a class of optimisations which can lower the energy consumption by reducing the k_P coefficient?
- Are these optimisations significantly different in terms of structure and application, when compared to standard optimisations?
- Can these optimisations effectively reduce energy?

Combinations of optimisations for time and optimisations for energy.

- Do optimisations designed to lower k_P change the efficacy of optimisations which lower k_T? Are there significant interactions between the two classes of optimisation?
- Are the same optimisations that are effective for a benchmark still effective in the presence of energy optimisations?
- Are different sets of optimisations found to be effective when including energy optimisations?

The answers to these questions will allow insight to be gained about the differences between optimisations for execution time and optimisations for energy consumption, and their composability. This will enable more effective energy optimisations to be designed to complement the optimisations already inside compilers.

This page is intentionally blank.

Chapter 3. Benchmarking

Work in this chapter also appears in the following publication:

- James Pallister, Simon J. Hollis and Jeremy Bennett. "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms". In: *arXiv, CoRR*. 2013.

Benchmarking is necessary to compare the efficacy of code across different configurations of the platform or compiler. For this thesis, benchmarking is needed to compare the execution time and energy consumption before and after applying code transformations to the program. Additionally, the benchmark suite must be representative of applications which would be run on the processor, and exercise the important aspects of the platform.

Many benchmark suites exist already, such as MiBench [46], MediaBench [47], LINPACK [48], Dhrystone [49] and more. These are all targeted towards larger desktop-based applications, with significant compute power, due to their emphasis on measuring performance, as opposed to energy efficiency. Most at least assume a host operating system is present, which may not be the case on an embedded system. Furthermore, when analysing energy consumption, having to account for the operating systems effect on the result is non-trivial. These benchmarks — while in theory are portable — have significant difficulties running unmodified on embedded platforms due to memory and storage constraints. The issue with portability also arises when it is necessary to contrast multiple platforms.

Prior to this work, a suitable benchmark suite for measuring energy consumption was not available for embedded platforms. While EEMBC have a recent ULPBench [50] product which could be ported to some of the platforms used in this thesis, it is not freely available and no details about the type of workloads in includes are given.MiBench is the closest to an energy-efficiency benchmark suite in terms of variety of benchmarks and applicability but assumes there is a host operating system for the majority of the benchmarks. In particular it requires access to a filesystem, which is usually unavailable on small embedded platforms.

Benchmarking for energy consumption is different to benchmarking for time. When benchmarking an optimisation for its execution time, a set of benchmarks where the operations (for example, the instructions) can take different times is necessary, whereas when benchmarking for energy, each operation should consume a range of energies. For example, accessing the flash and the RAM in an embedded system often requires the same number of cycles (and therefore time), however differs in energy consumption. Therefore, a benchmark suite designed to expose energy consumption artefacts should have benchmarks whose operations access flash and RAM in different ways, for example.

The actual measurement of energy itself is more challenging than time — all factors that affect time must be considered, as well as factors which change the power dissipation. Power is affected by the environment, and particularly by temperature. Also, the power measurement itself can be inherently imprecise, with noise from power supplies and nearby electronics interfering with the hardware. Chip-to-chip manufacturing variations can also impact the power dissipation. These do not generally affect the time measurement — typically the clock generator utilises an accurate crystal.

These variations may be small individually, but the effect of individual optimisations is also small; sometimes an optimisation only changes performance or energy by 1–2%. Therefore, the measurement setup needs to be accurate and multiple runs of the test performed to ensure

correctness.

This chapter discusses benchmarking in general, the measurement of the target metrics (such as energy and time), the choice of target SoC, and the construction the benchmark suite to be used for embedded SoCs. The suite forms the basis of the experimental work for the remainder of this thesis.

3.1. Background

Of the many existing benchmark suites, few target embedded systems, with most targeting either desktop machines (e.g. Dhrystone [49]) or HPC (e.g. PARSEC [51]). Few also explicitly target multithreaded systems, and none explicitly aim for energy as the target metric. This section discusses the relevant existing benchmark suites, which will be drawn upon to create a suite targeted for evaluating compiler optimisations and energy consumption.

MiBench [46] established a well known set of benchmarks with well characterised behaviour. This suite consisted of 37 different benchmarks split across six different categories, chosen to be representative of applications which would be run on both desktop and embedded platforms. The inclusion of each benchmark is justified, with instruction traces analysed on a model of the StrongARM architecture. This gave a good representation of the proportions of each type of instructions that the benchmarks executed. The drawback of the approach was that the instruction traces were only gathered for one platform — each benchmark could have a radically different instruction distribution for alternative platforms, possibly leading to some performance characteristics being overlooked.

MiBench was used as the main benchmark suite for the MILEPOST GCC study [52]. The study applied machine learning to predict which optimisations would benefit a program without needing to perform expensive iterative compilation techniques. In this study, they emphasised how the performance achieved can be very dependent on the structure of the benchmarks. This highlights the need to have a wide range of benchmarks which each hit different combinations of the types of computation they perform.

ParMiBench, a variant of MiBench was created to address the lack of multithreadedness in the original suite [53]. It attempts to parallelise some of the benchmarks, allowing multicore systems to be benchmarked. This has an advantage over other parallel benchmark suites in that it also targets the embedded space. Very few other benchmark suites (such as LINPACK, PARSEC and SPLASH-2 [51]) target multithreadedness at this level — most are aimed at large clusters and HPC applications.

Weiland et al. [54] develop parallel benchmarks with the aim of running on both embedded and high-performance systems, with a focus on power consumption. However, the use of embedded in this context is taken to mean mobile-phone-type application processors — significantly more powerful than the *deeply* embedded processors targeted in this thesis (with the exception of the AM335x, an application processor). Their embedded benchmarks utilise network and disk I/O, as well as caching — none of which are present on the platforms tested in this thesis.

DSPstone [55] is a benchmark suite for Digital Signal Processors (DSPs) and was originally designed to evaluate compiler effectiveness at compiling for DSPs. This suite contains many non-integer tests, with most tests replicated in fixed point and floating-point form. DSPstone is relevant since DSPs are often deeply embedded, however, they typically perform more floating-point operations than other embedded processors.

A set of benchmarks is maintained by the worst case execution time (WCET) initiative [56]. These benchmarks are appropriate for deeply embedded systems because they are self contained and written completely in C. Each benchmark is less comprehensive than its equivalent from the

MiBench set, but focuses on one particular application that may be specifically what a low-end processor will perform. Some of these applications fit well with typical embedded applications.

None of these benchmark suites explicitly cater to energy consumption, however, a suite to specifically target energy consumption is useful because of the differing energy costs of each instruction in a processor's instruction set. Many previous studies [57, 58, 12, 11] have attached an energy cost to each instruction and find that different instructions can have significantly different energies even if they take a similar amount of time. This justifies the need for a benchmark suite which is designed around these criteria, and for the measurement of energy consumption.

3.1.1. *Metrics*

Each metric of interest has a different measurement methodology and is useful in different situations. This section discusses some of the metrics used, and how they can be measured.

The average power and peak power metrics are useful for various platforms which have limits on the power supply (maximum power draw), or the amount of heat that the processor's package can dissipate. In general, power can be measured by instrumenting a processor with circuitry designed to periodically sample the instantaneous power draw (see Figure 3.1 for more information). The output of the power measurement circuitry is a sequence of power samples P_i, where $i = 0, 1, 2, ..., n$ and a time stamp for when each sample was taken, T_i.

Execution time is the typical performance metric, measuring how long a benchmark takes to complete. This metric is not necessarily meaningful if the application executes continuously, unless specific tasks should be executed faster so the processor can sleep (for example). The total execution time for a benchmark can be found from the same measurement hardware, as the largest (and final) time stamp,

$$T = \max(T_i) = T_n. \tag{3.1}$$

The energy consumption is an important metric, as it represents the amount of work necessary to perform the calculation in the benchmark. Lowering the energy consumption will increase the battery life of the device for a given battery capacity, resulting in devices which need to be recharged less frequently, use smaller batteries or have batteries replaced less frequently. The exact energy is given by the integral,

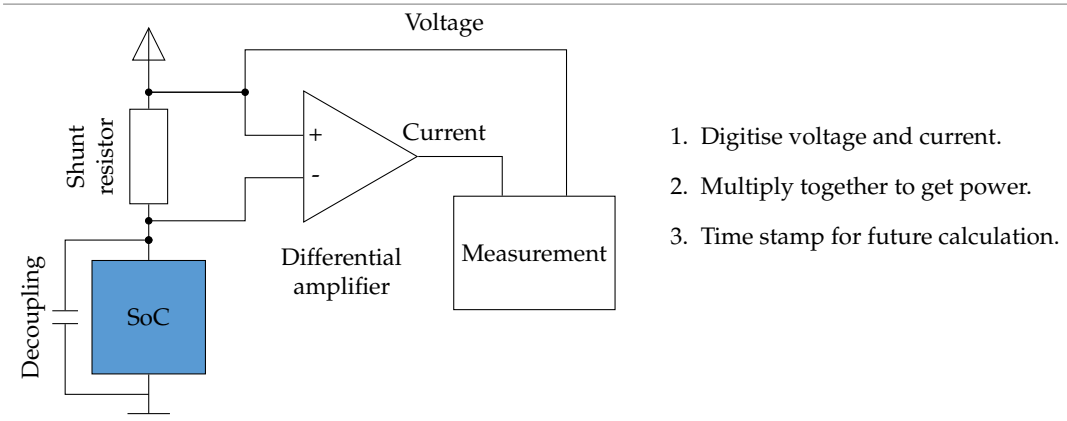


Figure 3.1: Power measurement setup.

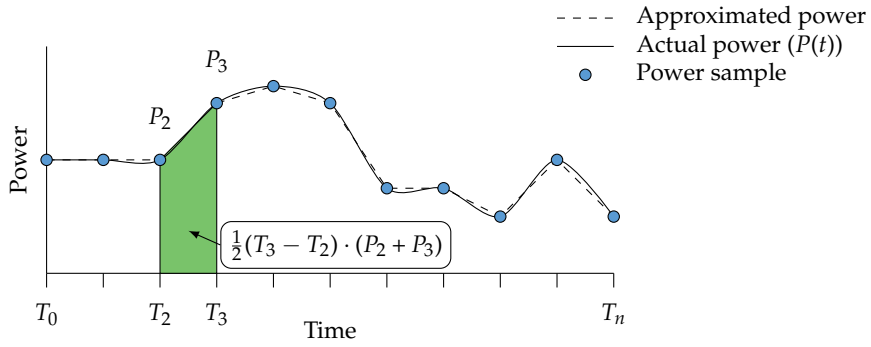


Figure 3.2: Example of integrating power and time to produce energy consumption.

$$E = \int_0^T P(t)\,\mathrm{d}t, \tag{3.2}$$

where $P(t)$ is the function of power with respect to time. Since the measurement is discrete, the energy consumption is estimated from the individual power samples,

$$E \approx \tfrac{1}{2} \sum_{i=0}^{n-1} (T_{i+1} - T_i) \cdot (P_i + P_{i+1}). \tag{3.3}$$

The formula uses each pair of points as a trapezium and sums the area of each trapezium. See Figure 3.2 for an example.

The average power can be found by dividing the total energy consumption by time,

$$P_{avg} = \frac{E}{T}\ , \tag{3.4}$$

where the time, T, is defined in Equation 3.1, and the energy, E, is defined in Equation 3.3. The average power can also be approximated by simply averaging all power samples if the period between each power sample is identical.

Other metrics have been proposed as a way to evaluate both energy consumption and time together, recognising that often it is necessary to achieve low energy consumption and short execution time together. The most commonly used metric of this type is the energy-delay product (EDP) [59, 60],

$$\mathrm{EDP} = E \cdot T. \tag{3.5}$$

The energy-delay product is a metric equally weighting time and energy and is useful in situations where there is a trade-off between energy and time, such as Dynamic Voltage and Frequency Scaling (DVFS). In some situations more weight needs to be given to either energy or execution time. In this case Laros et al. [61] propose weighting the delay factor (T),

$$E \cdot T^w, \tag{3.6}$$

where w is the weight of the execution time on the metric. Placing additional weight on the execution time is useful for applications with stricter deadlines, whereas weighting energy more heavily should be useful for energy-constrained devices.

An estimate of the peak power can be found by taking the largest instantaneous power recorded, P_i,

$$P_{peak} = \max(P_i). \tag{3.7}$$

The peak power is useful in some circumstances, since it affects the temperature of the device, and must stay within limits imposed by the power supply.

3.2. Measurement

The metrics of interest were measured using the MAGEEC [40] WAND, a board designed for taking energy measurements of small embedded platforms. The firmware and host-side software were designed by the author of this thesis. The board takes measurements of a benchmark running on the device-under-test and reports the energy consumption, execution time and average power dissipation back to the host.

As in Figure 3.1, the board takes two measurements, the voltage of the SoC and the voltage across the shunt resistor, and digitises them. Then, the figures can be integrated into an energy figure. The device-under-test has one more wire connecting to the measurement board — this is toggled when the benchmark starts and finishes, allowing the measurement board to record only during the execution of the benchmark.

Once a measurement has been taken, it is stored on the board until the host requests it. This allows the host to coordinate the execution of the benchmark on the device-under-test, and the energy measurement.

3.3. Platforms

Several SoCs are used in this thesis, representing a range of instruction sets, processor architectures and SoC configurations. Table 3.1 shows details about all of the platforms used. The majority utilise small, deeply embedded SoCs, with the AM335x (an application processor) and Epiphany (a DSP coprocessor) being used for more high performance applications.

These platforms are used in a variety of applications, from automotive [62] to satellites [63] and battery controllers [64]. Some of the simplest processors, such as the PIC32 and the ATMEGA are used in deeply embedded applications, often running on batteries which must last for years [65]. Others, such as the AM335x are used in higher performance applications, such as mobile phones [66]. The simple processors (top seven in the table) are all configured to use the same clock frequency (8MHz), even though some of them can run faster, to allow easier comparisons throughout this thesis.

3.4. BEEBS

As part of the contribution of this thesis, a benchmark suite was designed to run on the given embedded platforms, obeying the constraints imposed by most of the SoCs. This benchmark suite, BEEBS (Bristol/Embecosm Embedded Benchmark Suite), is designed fit within the memory constraints of most devices and provide a way of assessing compiler optimisations for deeply embedded SoCs.

The benchmarks in BEEBS are derived and adapted from a variety of other benchmark suites — MiBench [46], the Mälardalen WCET benchmarks [56] and DSPstone [55]. Benchmarks from

Mnemonic	Board Name	SoC Processor	RAM	ROM	Clock	Compiler GCC	 LLVM	Comments
STM32F0	STM32F0DISCOVERY	STM32F051 [69] **Cortex-M0**	SRAM 8kB	Flash 64kB	8MHz	✓	✓	A simple 3-stage pipeline, using the ARM Thumb instruction set. No caches.
STM32F1	STM32VLDISCOVERY	STM32F100 [70] **Cortex-M3**	SRAM 8kB	Flash 128kB	8MHz	✓	✓	A 3-stage pipeline, ARM Thumb/ThumbV2 instruction set, speculative branch execution. No caches.
ATMEGA	Breadboard	ATMEGA328P [71] **AVR8**	SRAM 2kB	Flash 32kB	8MHz	✓	×	An 8-bit processor, with separate program and data memory spaces. No caches.
XMEGA	XMEGA-A3BU Xplained	ATXMEGA256A3BU [72] **AVR8**	SRAM 16kB	Flash 256kB	8MHz	✓	×	An 8-bit processor, with separate program and data memory spaces. No caches.
PIC32	Breadboard	PIC32MX250F128B[73] **MIPS32**	SRAM 32kB	Flash 128kB	8MHz	✓	×	A standard MIPS32 core. No caches.
MSP430F	MSP-EXP430F5229LP	MSP430F5529 [74] **MSP430**	SRAM 10kB	Flash 128kB	8MHz	✓	×	An 16-bit processor, with a RISC-like instruction set, but more complex addressing modes. No caches.
MSP430FR	MSP-EXP430FR5739LP	MSP430FR5739 [75] **MSP430**	SRAM 1kB	FRAM 16kB	8MHz	✓	×	Identical to above, however with ferro-electric RAM instead of flash [76]. No caches.
AM335x	BeagleBone	AM335x [77, 78] **Cortex-A8**	DRAM 256MB	SD-Card	800MHz	✓	✓	32kB L1 cache and 512kB L2 cache, ARM mode instruction set, NEON SIMD unit.
Epiphany	EMEK4 Dev Board	EMEK4 [79] **Epiphany**	SRAM 32kB†	None	800MHz	✓	×	16 RISC-like cores with floating-point, connected by three mesh networks. No caches.
XMOS	XK-1A	XS1-L8-64 [80] **XMOS-XS1**	SRAM 64k	None‡	400MHz	×	✓	Up to eight hardware threads, executed in a round-robin fashion. No caches.

Table 3.1: The platforms used in this thesis.

† Local memory. Also 512MB external DRAM.
‡ Off chip flash is attached as ROM.

many other suites were also evaluated [47, 67, 48, 68], however, the majority require operating system support not found on embedded processors. Related individual benchmarks were combined to represent typical application usage (such as an application both encrypting and decrypting data) and the benchmarks refactored to remove I/O operations. This is necessary due to inconsistent methods of I/O on each platform.

The benchmarks were chosen so that they cover a wide variety of different operations that may affect program execution metrics, such as energy and time. The characteristics evaluated were:

Integer operation intensity. Integer operations are typically the cheapest operation a processor can perform, both in terms of energy and time. Integer operations typically only stress the ALU and the processor pipeline.

Floating-point operation intensity. Floating-point operations are more specialised than integers, but are still frequently used for certain applications. These operations stress the FPU, which requires much more silicon area than an integer ALU. Therefore, these operations can have greater latency and power requirements.

Branch frequency. The amount of branching a benchmark performs will greatly impact the execution time and energy consumption of a benchmark, due to pipeline flushes, and other instruction prefetching hardware having an effect. Branch frequency also determines how much certain components involving large areas of silicon are used, such as caching and branch predictors.

Memory access frequency. In many deeply embedded systems, a memory access can be performed in a single cycle. However, that memory access typically requires much more energy than a register access. By contrast in more performant SoCs, the memory access pattern and bandwidth required exercises the caches and memories, as well as stalling the processor if the request cannot be fulfilled quickly.

Ensuring that each benchmark covers a different area of this four-dimensional space allows the suite to expose the energy and performance characteristics of embedded processors.

The benchmarks are also selected to cover several application categories. Six categories are proposed by MiBench, of which five cover most application areas for embedded programs. The final category — office applications — are infrequently seen on deeply embedded devices and the majority of benchmarks in this category were running off-the-shelf programs requiring Linux.

Automotive. These applications typically control devices and sensors, such as engine control and sensor data processing. As such, they utilise arithmetic abilities and data manipulation, and algorithms involving filters, matrices and linear algebra.

Consumer. Consumer applications are typically restricted to more powerful systems, however, some embedded systems perform various audio and visual decoding and transformations, such as video playback.

Network. Many embedded processors implement network protocols to handle communication. Devices such as routers and wireless sensor nodes need to perform various checks on the data and graph operations on the network structure.

Security. The security category involves the hashing of data, as well as the signing and encryption of data for cryptographic purposes. These algorithms infrequently use floating-point numbers, instead requiring high integer performance.

Telecom. Telecom applications involve radio frequency signals and data analysis. Similarly to network, the telecoms category includes protocol encoding and decoding.

Since the compilation of the benchmarks and the large range of compiler optimisations will greatly affect their efficiency, there should also be a range of features in the benchmarks which affect the compilation process. These include loops, nested loops, different arithmetic types (8-bit, 16-bit and 32-bit integers, float, etc.), calls to functions, string operations, bitwise operations and array accesses. These features are similar to those suggested in the WCET benchmarks [56], where they are chosen because worst case execution time tools must be able to cope with these structures.

A benchmark suite which covers this range of application categories, has a range of benchmarks with different instruction characteristics, and obeys the constraints of an embedded system is suitable for evaluating energy efficiency. With the exception of two platforms (of the list in Table 3.1), the benchmarks will fit in the memory of all platforms, and run on all SoCs. The two platforms which cannot run some of the benchmarks are ATMEGA and MSP430FR, due to extremely limited memory.

3.4.1. *Benchmarks*

This thesis chooses ten benchmarks for the BEEBS benchmark suite, covering a range of the criteria and application categories.

Name	Description
2dfir Two-dimensional FIR filter	Finite Impulse Response (FIR) filters are frequently used in image transformations. In the embedded space, this could be the type of operations done by digital cameras. This benchmark is similar to the matrix multiplications but with potentially more memory accesses and spatially different operand sources for arithmetic.
blowfish Blowfish encryption	Blowfish is an encryption algorithm commonly used in cryptography. This benchmark was taken from MiBench but modified to both encrypt and decrypt small blocks of data, as if the data was being streamed into and out of the processor. The stream is generated pseudo-randomly to avoid platform dependencies on input and output. Encryption typically involves many integer operations with fewer, predictable branches.
crc32 32-bit CRC	CRC32 (32-bit cyclic redundancy check) is commonly used for verification of data streams, notably ethernet frames. It can be implemented with very few instructions as it consists mainly of shifts and XORs. As it consists of few instructions in a tight loop, this benchmark should exercise processors with superscalar execution or branch prediction. The benchmark performs the CRC on a stream of pseudo-randomly generated data.
cubic Cubic root solver	This benchmark performs a large amount of trigonometry to solve various cubic equations. This tests the floating-point pipeline with very little memory required. This is a portion of the "basicmath" benchmark in MiBench, cut down to fit on smaller processors (with limited RAM).

Name	Description
dijkstra Shortest path using Dijkstra's algorithm	This benchmark implements the Dijkstra shortest path algorithm, performing frequent non-linear accesses to memory, and branching unpredictably. This makes it good for stressing caches and any branch predictors that the processor may have. The algorithm is commonly used by routers to calculate the shortest path to another router. This benchmark was modified from the MiBench version to have the adjacency matrix embedded in the source code, rather than loaded from the filesystem.
fdct Finite Discrete Cosine Transform	The Finite Discrete Cosine Transform (FDCT) benchmark was included as it is a core algorithm behind many video decoders used in consumer products. This benchmark represents real-world usage of the systems as well as testing the floating-point pipeline and caches.
matmult-int Integer matrix multiply	Integer matrix multiplication is used very frequently in many applications, and so is a useful benchmark to have. It consists of a tight inner loop with many array accesses, making it useful for stressing the memory and integer pipeline at the same time. This should also expose data caching effects of the platform.
matmult-float Floating-point matrix multiply	Floating-point matrix multiplication is also used frequently in many applications. This benchmark is a modified version of the integer matrix multiplication benchmark, with floating-point numbers in place of integer — all other code is identical. This should allow a good metric of relative performance between the integer and floating-point pipeline to be produced.
rijndael AES encryption	Rijndael is the algorithm for the Advanced Encryption Standard (AES). It is commonly used in many security applications, and has a similar structure to blowfish. It also has similar execution characteristics except for more frequent branching.
sha Secure Hashing Algorithm	Secure Hashing Algorithm (SHA) is commonly used for fingerprinting and verification of data streams. It is useful for stressing integer pipelines, and has low memory requirements. The benchmark hashes a stream of pseudo randomly generated data.

3.4.2. *Evaluation*

This section provides a concrete analysis of all the chosen benchmarks by collecting their instruction traces across a subset of the available platforms, with the aim of verifying a good distribution of instruction types was achieved for different SoCs. The STM32F0, XMOS, Epiphany, and ATMEGA SoCs were chosen due to their differing characteristics, including different memory sizes, pipeline depths, instruction set features (conditional execution, floating-point, etc.) and data-path bit-widths. The instructions can be categorised to demonstrate that each benchmark performed a different distribution of operations. Figure 3.3 shows the instruction distribution for the STM32F0 (ARM Cortex-M0, Thumb instruction set), XMOS, Epiphany, and ATMEGA (AVR) platforms. The 'Other' category of instructions contains miscellaneous control instructions that do not fit into other categories (for example, interrupt control on the Epiphany platform). It

Type	Benchmark distribution ranges (%)			
	Epiphany	XMOS	STM32F0	ATMEGA
Integer	26–77	28–68	37–79	60–82
Floating-point	0–49	–	–	–
Memory	10–30	17–43	6–34	3–30
Branching	1–20	1–30	1–42	10–26

Table 3.3: Instruction distributions dependency on the platforms and the benchmarks.

was not possible to adapt three benchmarks for the ATMEGA SoC — *blowfish*, *matmult-int* and *rijndael*. These benchmarks required more than the 2kB of available memory and could not be run on the device.

The integer instruction category is the largest group in almost every case, for all platforms and benchmarks, and the distributions are similar, with small variations due to the underlying instruction set. This comes from the integer category covering the largest number of types of instructions, as it groups arithmetic, register copying and bit-wise operations. For example, there is a larger percentage of `mov`-type instructions in the Epiphany results because there are several predicated `mov` instructions (`moveq`, `movlt`, etc). This reduces the need for conditional branches, so this category decreases in proportion. The ATMEGA executes integer operations much more frequently than any other type — many benchmarks utilise 16- or 32-bit arithmetic, which must be emulated by the 8-bit processor.

Epiphany is the only platform in the subset chosen which has hardware support for floating-point. For the other platforms, software library calls are inserted in place of the operations by the compiler. On the XMOS platform this manifests in an increased proportion of branch and memory instructions, whereas for the STM32F0 platform the proportion of integer operations rises. These differences are due to the different emulation strategies used. The STM32F0 traces follow the same general trend as the traces for XMOS and Epiphany, however with overall fewer memory operations, since the Cortex-M0 processor has support for the `ldm` and `stm` instruction allowing multiple accesses to memory in a single instruction. These instructions are used extensively in function prologues and epilogues to save and restore registers. Floating-point instructions appear in the traces for benchmarks which do not use floating-point types in the Epiphany processor — the floating-point pipeline can be reconfigured to perform integer operations instead.

3.4.3. *Summary*

These benchmarks show a range of different quantities of each instruction class, with similarities across platforms. This makes the set of benchmarks ideal for use in energy profiling a system. For all platforms a given benchmark produces a similar instruction profile. This is shown in Table 3.3. The columns in the table show the overall range of instructions for the benchmark suite. These ranges are similar for all the platforms, with the most significant difference being the ATMEGA, with at least 60% of its instructions being integer. This due to the 8-bit nature of the processor requiring it to perform multiple instructions to emulate longer bit widths. Between benchmarks there is significant variation in instruction distributions, therefore the suite explores a wide range of input configurations in a consistent manner between architectures.

Overall the benchmark suite covers a wide range of applications in several real-world application categories. The instruction distribution differs between each benchmark and provides a good coverage of each instruction type, even across different SoCs. Along with a range of features which affect the compilation of the benchmarks, this makes the suite ideal for both

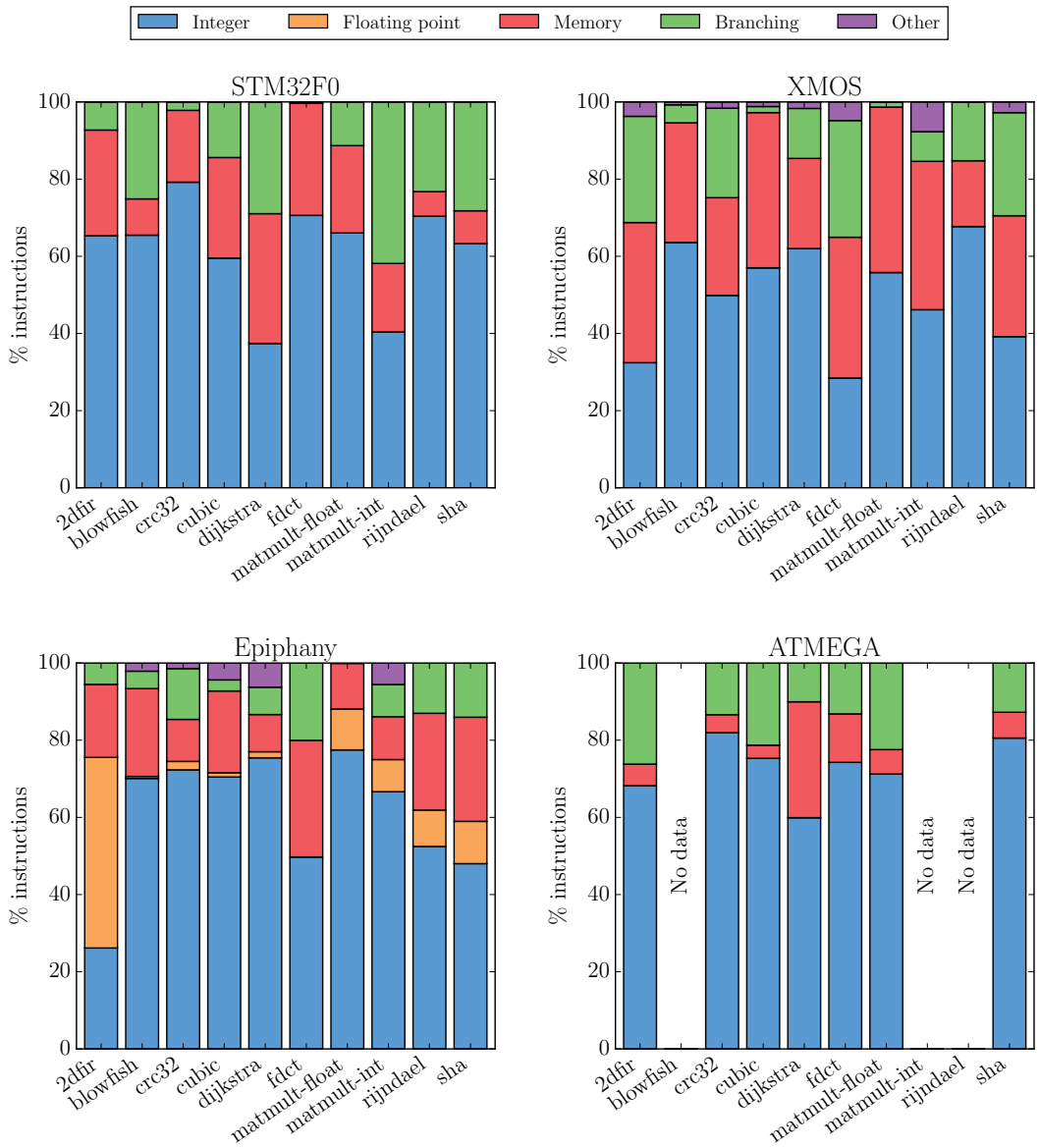


Figure 3.3: Instruction distributions for different SoCs.

examining compiler optimisations and evaluating energy consumption for embedded platforms. BEEBS is therefore used as the standard benchmark suite during this thesis.

Chapter 4.
Optimisations designed for execution time

Work in this chapter also appears in the following publication:

- James Pallister, Simon J. Hollis and Jeremy Bennett. "Identifying compiler options to minimize energy consumption for embedded platforms". In: *The Computer Journal* 58.1 (Nov. 2013), pp. 95–109.

This chapter explores the existing optimisation in the compiler (GCC), and their effect on both energy and time.

4.1. Introduction

There are many compiler optimisations in existence, ranging from optimisations which attempt to remove unneeded computation by refactoring the code, to optimisations which attempt to reorder instructions using a model of the processor's pipeline. Each optimisation has its own criteria for application, and effect upon the source code. In general, the most effective sequence of optimisations to apply is not obvious, therefore current compilers typically have predefined sequences of optimisations, grouped into an *optimisation level*. These are combinations of optimisations in a specific order, which have been determined by the compiler writer as an acceptable trade-off between compilation time and performance of the compiled code. As such, this predefined combination almost never reaches optimal performance.

Most of the optimisations existing in current compilers are designed for performance: instruction latencies are used to schedule, register allocation attempts to reduce the number of spill instructions and instruction selection prioritises the shortest execution time. The effect on a program's execution time can be large — achieving a 50% reduction in execution time by enabling optimisation is not uncommon for many applications. The lower execution time also helps embedded systems with their goal of lower energy, since frequently these optimisations reduce the total amount of work to be performed (i.e. a lower k_T constant, as discussed in Chapter 2). Therefore, it is useful to explore the effects of the existing optimisations on the energy consumption before creating new optimisations specifically targeted at energy.

This chapter explores how existing optimisations in the compiler (GCC [41]) affect energy consumption, and how the selection of these optimisations affects the energy consumption. First, background on combining optimisations and optimisation selection is presented. Then, an analysis of the main optimisation groups in the compiler is given. The optimisations inside each group are then explored for their individual effect and efficacy across the BEEBS benchmarks and the platforms. Finally, a genetic algorithm is used to attempt to select an effective set of optimisations, and find cases where energy can be saved at the expense of execution time.

4.1.1. *Optimisation combinations*

The exact selection or sequence of optimisations applied to a program can have a huge effect on the final performance of the program, sometimes with orders of magnitude difference. However, the optimal choice of optimisations is difficult to determine, due to the large number of interactions between optimisations. The effect of ordering different optimisations and

interactions between optimisations is demonstrated with the example in Figure 4.1. This example considers the interaction between *common subexpression elimination* (CSE[1], see Appendix A, page 99 for more details) and *function inlining* (see page 101). With just these two optimisations, the following four cases are discussed:

CSE only. If only CSE is performed, the call to the optimiser discovers multiple calls to `function1` with identical arguments and saves the result in a temporary, eliminating one of the calls[2] (Figure 4.1). The temporary is used for the final multiplication, resulting in a smaller code size and faster execution — less work needs to be repeated.

Inlining only. If this code snippet is compiled with function inlining, then `function1` can be identified as a small, inlinable function, and will be substituted into `function2`, eliminating the function call overhead.

CSE then inlining. With CSE performed first, the function inlining affects the temporary, resulting in two multiplies and two additions.

Inlining then CSE. When CSE is applied after inlining, the optimiser will discover that the expression `(b + 1)` appears multiple times, and assign it to a temporary variable. It then replaces the expressions with this temporary variable, resulting in the code in the bottom-right of Figure 4.1. This expression minimises the number of additions that need to be performed (three multiplications, one addition), usually resulting in smaller and faster code — the total work that must be performed is lower.

In each case the resulting code is different and will have differing performance and energy consumption characteristics. Note that for both cases with CSE and inlining performed, CSE can be applied again and will further reduce the amount of code, giving:

```
int function2(int b)
{
    int t1 = b + 1;
    int t2 = t1 * t1
    return t2 * t2;
}
```

Apart from temporary names, both CSE–inlining–CSE and inlining–CSE–CSE result in the same final program, although this is not necessarily the case for other programs or optimisations. Neither CSE nor inlining can be applied further to the program. In a typical compiler, CSE will be applied repeatedly until there are no more expressions to be eliminated.

The applicability of an optimisation to the source code is determined by the structure of the source code itself. In general, an optimisation will search for a particular pattern in the Abstract Syntax Tree (AST) or Intermediary Representation (IR) and, when found, perform a transformation. This modifies the AST or IR each time an optimisation is applied, resulting in a different set of optimisations which can be applied subsequently. This causes complexity in choosing sequences of optimisations — the applicability of a sequence depends on the exact structure of the input program. Considering the sequencing of optimisations allows modern compilers to be split into two categories:

- Compilers which apply their optimisations in a fixed order, and individual optimisations are just enabled or disabled in this sequence.

[1] In this example, CSE is not applied until there are no common subexpressions remaining, so may be termed *partial* common subexpression elimination.

[2] The optimisation can be applied because `function1` has no side effects.

No transformations

```
int function1(int a)
{
    return a * a;
}

int function2(int b)
{
    return function1(b + 1)
             * function1(b + 1);
}
```

CSE

```
int function1(int a)
{
    return a * a;
}

int function2(int b)
{
    int t1 = function1(b + 1);
    return t1 * t1;
}
```

Inlining

```
int function2(int b)
{
    return ((b + 1) * (b + 1))
             * ((b + 1) * (b + 1));
}
```

CSE, then inlining

```
int function2(int b)
{
    int t1 = (b + 1) * (b + 1);
    return t1 * t1;
}
```

Inlining, then CSE

```
int function2(int b)
{
    int t1 = b + 1;
    return (t1 * t1) * (t1 * t1);
}
```

Figure 4.1: Source code representation of the effect of applying different orderings of optimisations.

- Compilers which allow optimisations to be applied an arbitrary order.

GCC 4.8 [41] is an example of the first type — many optimisations exist but their order is fixed and cannot be changed without modifying the compiler. On the other hand, LLVM 3.5 [42] allows most optimisations to be applied in any order — possibly specified by the developer. By allowing complete freedom in specifying the order of the transformations the compiler can potentially create faster code, however, the ordering of the compiler optimisations is a difficult problem, the "phase ordering problem" [81].

4.2. Background

Many studies have considered how to select and order optimisations to increase performance or decrease energy consumption. These techniques range from iterative compilation to genetic algorithms and machine learning to choose optimisations. Further studies have analysed the effect of individual optimisations on energy consumption. These are now explored in detail.

4.2.1. *Optimisation selection*

Pan et al. [82] explore several approaches to optimisation selection (for performance), and propose a new method based on several state-of-the-art techniques. *Batch Elimination* executes the application several times each with one optimisation disabled, then assumes the best set is the set with all performance-reducing optimisations disabled. However, this approach does not always perform well, since it does not consider the interactions between optimisations. Another algorithm, *Iterative Elimination*, tries to iteratively find optimisations to eliminate by testing optimisations individually and turning off the one with the largest negative impact. The process is repeated until no optimisations with negative impacts are left — this algorithm is very similar to a restricted form of hill climbing [83].

Pan et al. combine the two algorithms, attempting to iteratively eliminate multiple optimisations and achieving a lower number of tests required than iterative elimination, while obtaining a similar performance. However, since these algorithms are fairly restrictive in terms of the set of optimisations they consider, performance increases of only 5–10% were seen.

Several studies explore the optimisation space in a more systematic way, allowing general conclusions about the nature of the space to be made. Iterative compilation has been used to explore the loop tile size and unroll factor parameters in the compiler in [9]. The effect on performance, energy and the energy delay product is shown to have a recurring pattern in the 2D space of tile size and unroll factor. They conclude that iterative compilation is a promising approach that may reduce energy when used on a larger number of loop transformations and combinations of optimisations. The majority of improvement in energy consumption in this case comes from the increase in performance, and requires a large number of compilations and tests to find a good set of parameters.

Chow et al. [84] take the approach of using fractional factorial design [85], choosing a subset of tests to explore the effect of nine optimisations on an application's performance. Generally, fractional factorial design is an experimental-design methodology which allows a large number of possibly interacting factors to be explored systematically (see Section 4.4.1) [85]. Fractional factorial design reduced the number of runs to 32, instead of $2^9 = 512$, allowing the simulations of the different configurations to be completed in an acceptable time. Using the results derived from the fractional factorial design runs, a better set of optimisations was able to be chosen. The set improved the performance over naïvely selecting every optimisation, by choosing optimisations which had a significant effect, and considering the interactions between optimisations. A further conclusion of this study was that examining optimisations individually is a poor metric of how an optimisation would perform when combined with other optimisations.

Fractional factorial design was also applied iteratively to select the most effective optimisation at reducing energy [1]. A statistical test was used to choose the optimisation that reduced the energy consumption of the benchmark the most, while using fractional factorial design minimised the number of runs that needed to be performed. Using this technique they managed to reduce the energy consumption by up to 15% more than the highest optimisation level. This study used a larger range of optimisations[3] than most other studies, exploring 31 different optimisations. However, the study was only performed on a single platform.

4.2.2. *Machine learning*

The majority of studies conclude that the optimisation selection space is very difficult to explore in a way that allows a good set of optimisations to be chosen. Several papers have tackled this by using machine learning to generate a set of optimisations. Genetic algorithms

[3] However, the number of optimisations was still fewer than half the number available in the compiler.

have been used to choose which optimisations should be selected, by Lin et al. [86]. The study compared a basic genetic algorithm to a modification which attached weights to each of the genes (optimisations) within each individual. The weights were used to find groups of optimisations that are positively correlated and ensure they are all enabled or disabled for a particular number of generations. This modification allowed the genetic algorithm to find a better solution, and converge upon that solution faster, achieving up to 24% over the fastest optimisation level.

Cooper et al. [87] attempt to use genetic algorithms to reduce the code size of an application, by selecting and ordering compiler optimisations. The genetic algorithm managed to find benchmark-specific optimisation sequences which significantly reduced the code size, as well as finding a single optimisation sequence which performed within 6% of the best individual sequence (for code size). The optimisation sequences found sometimes made the program execute faster, suggesting that the optimisation sequence was managing to simplify the code sequence. In other cases the execution time was not affected (unreachable code elimination type optimisations) or was slower, ensuring optimisations which made the code faster but larger were not used (such as function inlining, or techniques outlined in [88]). Another study attempted to use genetic algorithms to select loop transformations, allowing a loop to be vectorised, and increase the performance [89].

Few studies attempt to minimise energy consumption using compiler optimisations with a genetic algorithm [90]. However, Schulte et al. [91] applied a genetic algorithm to the code generated by the compiler, attempting to minimise the energy consumption (using an energy model). The technique allowed the energy to be reduced significantly, although the amount of functionality retained by the program was not guaranteed or verified. In the extreme case, this could result in the algorithm removing all of the functionality and reducing the energy consumption to zero, however this was avoided by adding functionality to the fitness function.

MILEPOST [52] used machine learning to select a set of optimisations to apply to each benchmark in a suite based on static features of that benchmark. This involved a training phase, where sets of optimisations were applied to benchmarks and stored in a database so that the compiler could later make predictions when given a new benchmark. This allowed the compiler to achieve both code size and performance improvements without having to compile and test the benchmark iteratively.

Similarly to the MILEPOST GCC study, Cavazos et al. [92] used features of each benchmark to predict which optimisations would be beneficial. However, dynamic features were used in addition to statically analysing the benchmark, providing the compiler with runtime feedback on how effective the optimisations were and what the application was doing. To gather this data, hardware counters were used. They were able to achieve a 17% improvement in performance with only 2 additional runs for a particular benchmark. The use of the counters was compared to a pure random selection of optimisations. Due to the irregular nature of the optimisation-selection space (many minima close to the global minima), random selection performs well, but the prediction method achieves the same level of performance in 60 iterations of tests and refinement, compared to 200 (for random selection).

4.2.3. *Optimisation ordering*

Careful optimisation selection has been shown to both increase performance and decrease energy consumption over blindly applying every optimisation available. However, by considering the ordering of optimisations much greater gains can be gained. No known studies have focused on the effect of optimisation ordering on energy usage, however this area has been explored for performance. A review of heuristics used to explore this larger space has been undertaken by Kulkarni et al. [81]. This study looked at hill climbing, simulated annealing, genetic algorithms, random search and N-lookahead. A major finding is that the search space is highly irregular,

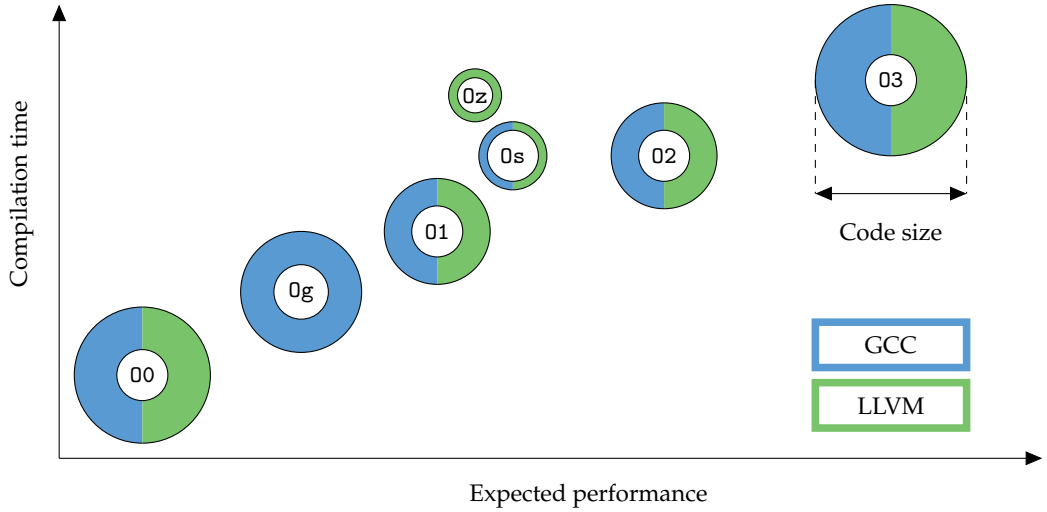


Figure 4.2: Illustration of the design goals of each optimisation level. The optimisation levels trade-off between compilation time, expected performance and code size (size of the circle).

with a few global minima, and that increasing the length of the optimisation sequence increases the number of global minima in the space.

As with optimisation selection, machine learning has also been used in the ordering of optimisations [93] based on features from the application being compiled. This approach used Neuro-Evolution for Augmenting Topologies (NEAT) [94] to learn the structure of an artificial neural network that would recognise a program's features and identify the next optimisation pass that should be run. A speed up of up to 20% over the best optimisation level was achieved.

4.2.4. *Individual optimisations*

Individual optimisations have also been extensively analysed for their effect on energy consumption. In addition to the loop optimisations (tiling, unrolling, see Appendix A) studied with iterative compilation by Gheorghita et al. [9] (described earlier), the energy impact of loop fusion was explored in [95]. The study used Dynamic Voltage and Frequency Scaling (DVFS) to reduce the energy consumption of an application. Loop fusion was then also applied and shown to reduce the performance impact of DVFS, while increasing the energy savings of the technique.

Single Instruction Multiple Data (SIMD) usage has been analysed with regards to power dissipation and energy consumption [10]. Overall, using SIMD instructions to parallelise code reduces energy consumption while increasing power dissipation, due to larger amounts of silicon area being active simultaneously. The majority of the energy reduction comes from the reduced runtime, as well as lower instruction overhead (fewer total instructions). This study also looked at some individual optimisations, finding that most optimisations either reduced or increased energy and time proportionally (i.e. the average power was unchanged). A few optimisations did affect average power dissipation: function inlining was found to reduce the power. Inlining can remove memory references which typically have a higher than average power dissipation. Loop unswitching — creating an extra loop by moving a conditional expression out of the loop body (see page 104) — was found to increase the average power, although no explanation was given.

A common technique to avoid the overhead of caches is to use a scratchpad memory. Scratch-

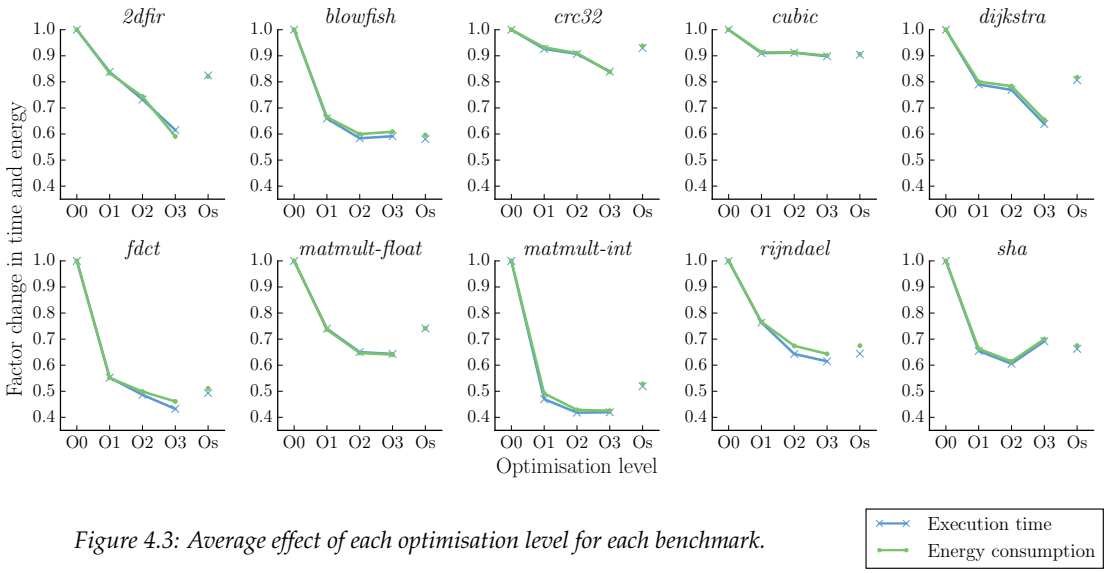


Figure 4.3: Average effect of each optimisation level for each benchmark.

pad memories are areas of fast RAM — often directly on chip — which are managed by the application, rather than by integrated hardware controls. While the memories are fast and simple, the complexity of utilising the scratchpad is shifted to the application or compiler. Since scratchpad memories are simple they consume less energy than caches. Several studies have examined how code or data can be moved into these memories to speed up memory access [96]. These studies also look at the energy consumption of the techniques, finding that up to 30% of the energy could be saved, with a performance increase of 25%. Ishitobi et al. [97] examine the case where both caches and scratchpad memories can be used, finding that 23% reduction in energy consumption can be achieved with no loss of performance.

Ortiz et al. [98] investigated the effect of several source code optimisations on power dissipation. The study used Analysis of Variance (ANOVA) to determine which optimisations had a significant impact on the power dissipation on three processors, finding that loop unrolling had an impact on two of the platforms, and the variable's data-type had an effect on one of the platforms (although the paper lacks information on exactly what this optimisation did). This study looked purely at power dissipation, ignoring the effect of the optimisation on execution time and energy consumption, making it difficult to draw conclusions about the efficacy of each optimisation.

Most of the studies described here achieve their energy savings in majority from performance improvement. Savings in energy are also increased by reductions in memory operations, although the additional benefit is low compared to the speed up.

4.3. Optimisation levels

Modern compilers have many optimisation passes — far too many to be individually controlled by the user. Therefore, compilers expose several combinations of optimisations, providing different trade-offs between performance, compilation time and code size. These optimisation levels are chosen based on the compiler writer's experience, both the selection and ordering. Some of the common optimisation levels are given below, and shown in Figure 4.2.

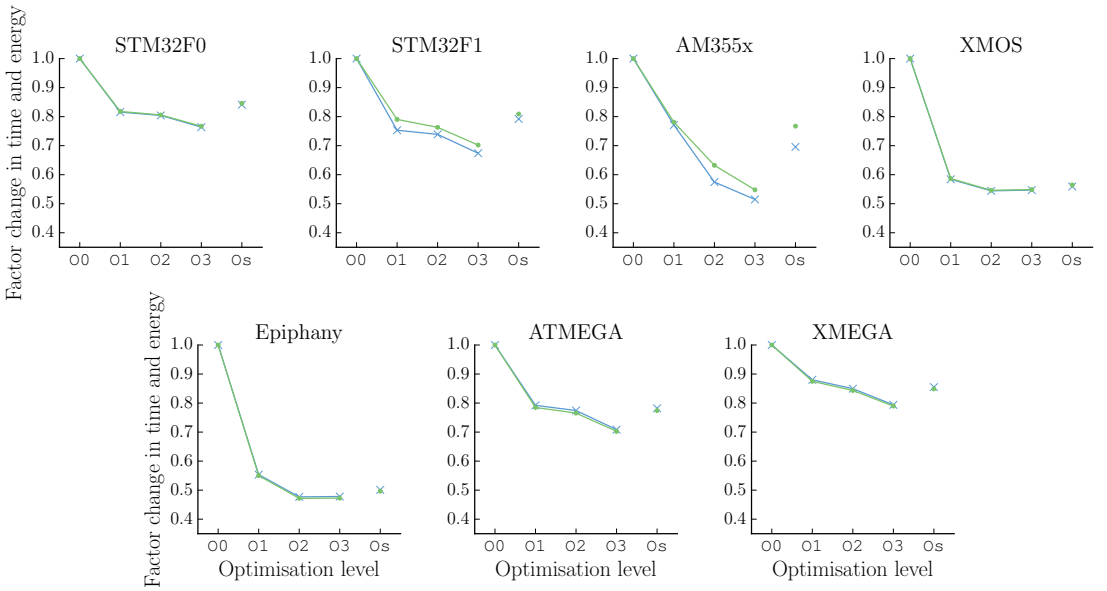


Figure 4.4: Average effect of each optimisation level for each platform over all benchmarks.

`O0` Do not apply any optimisations. This usually results in the least efficient code, however the compilation time is small. Additionally, the generated code maintains almost a one-to-one mapping with the source code, making it easy to debug.

`Og` Newer versions of GCC (above 4.9) have this grouping of flags which attempts to retain good debug information about the code — which is often destroyed when applying some optimisations — while still optimising the code

`O1` Apply a small set of basic optimisations. Optimisations which almost always have a positive effect on execution time and code size are applied here.

`O2` Apply a larger set of optimisations, including more complex optimisations such as instruction scheduling. While some of these optimisations reduce code size, others such as alignment of functions and labels will increase it.

`Os` Attempt to compile for code size. The set of optimisations used is very similar to `O2`, but with optimisations which increase code size disabled.

`Oz` Optimise for code size even more by doing optimisations which will likely reduce the performance of the program. This optimisation level is available in newer versions of LLVM.

`O3` Use sophisticated optimisations which take significant amounts of time to perform and may not necessarily improve performance. Optimisations such as unrolling and function inlining are enabled by GCC at this level.

Figure 4.3 shows the energy consumption and execution time effect of each optimisation level for each benchmark, averaged across platforms (for GCC). The overall trend is as expected

— increasing the optimisation level decreases both energy and time. However, the size of the reduction and efficacy of each optimisation level varies by benchmark. Some benchmarks, such as *cubic* do not see a large effect from applying optimisations. The lack of effective optimisation is limited by the structure of the code in *cubic*, as a result of many library calls which are resolved at link-time and not optimised.

On the other hand, the *sha* benchmark can be heavily optimised across all platforms. There are few library calls in this benchmark, and many structures in the code that the optimiser manages to exploit.

One benchmark, *2dfir*, sees the energy consumption decrease by more than the execution time. This is heavily skewed by the AM335x for this benchmark. The optimisations applied at `O3` for this benchmark applies a transformation which greatly reduces the amount of memory traffic and vectorises the code. Fewer costly off-chip accesses are necessary, reducing the overall energy.

The majority of benchmarks exhibit slightly different behaviour on different platforms due to the different instruction sets. The *2dfir* benchmark does not get significantly optimised until link-time optimisation is enabled in the STM32F0 and STM32F1, however the benchmark is effectively optimised at `O1` and `O2` for Epiphany and AM335x respectively. The average optimisation level effect for each SoC is seen in Figure 4.4. The general trend shows a processor with a complex pipeline benefits from higher optimisation levels, e.g. scheduling only has an effect when the pipeline is non-trivial. The STM32F0 (Cortex-M0), and XMOS processors both have simple and predictable pipelines, resulting in the higher optimisation levels having less effect, and thus less scope overall for optimisation. The Epiphany and AM335x (Cortex-A8) both have in-order superscalar processors and in general the optimisations have a larger effect.

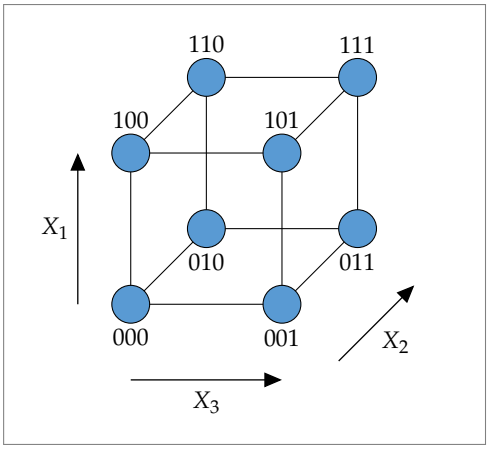
The AM335x has the most complex pipeline, and for this platform larger deviations between energy consumption and execution time are seen. For many of the benchmarks (*blowfish*, *dijkstra*, *fdct*, *matmult-int*, *rijndael*) execution time decreases by a larger proportion than energy consumption. Instruction scheduling and other optimisations which allow both pipelines of the processor to be utilised simultaneously enable this difference. The execution time is lowered since multiple instructions happen at once, but the energy is not lowered by as much, since a larger amount of the processor is active simultaneously, triggering higher power dissipation.

The STM32F1 sees a small divergence in energy consumption and execution time for some benchmarks (Figure 4.4). This is largely caused by scheduling eliminating pipeline stalls, and load/store pipelining inside the Cortex-M3. This feature allows consecutive loads and stores to be pipelined, with each subsequent operation taking only one additional cycle instead of two. While this reduces the instruction latency, the same amount of work must still be performed by the processor, so the energy consumption is not reduced by as much as the time.

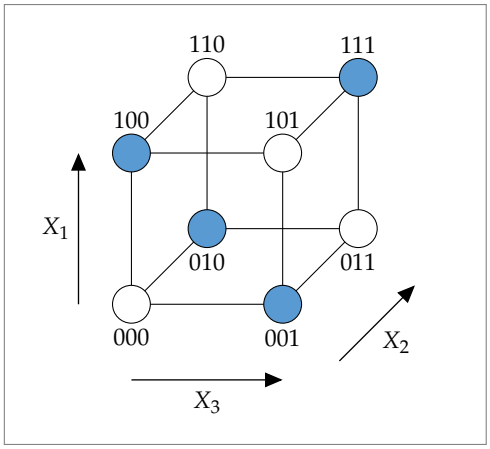
Overall, all of the benchmarks have different responses to the optimisation levels and in some cases energy diverges from execution time, although not by more than 10%. This suggests that the optimisation efficacy is dependent on the structure of the source code. Each of the platforms also has a different response to changing optimisation levels, with more complex processors having a much larger scope for effective optimisation. For most of the SoCs the energy reduction is proportional to the reduction in execution time — the bulk of the energy improvement is from faster execution.

4.4. Individual optimisation exploration

This section discusses how the selection of optimisations can be analysed. Compilers, such as GCC, apply their optimisations in a fixed order, however optimisations can be turned on and off



(a) Full factorial design of three optimisations, X_1, X_2 and X_3.



(b) Fractional factorial design of the three optimisations.

Figure 4.5: Full and fractional factorial designs for three optimisations.

individually. All of the optimisations available in GCC are analysed using fractional factorial design across five of the platforms, STM32F0, STM32F1, AM335x, Epiphany, and XMEGA.

4.4.1. *Fractional factorial design*

Fractional factorial design (FFD) is an experimental-design technique used to systematically explore a large combinatoric search space. Using this method, a reduced set of tests can be created to explore the efficacy of a set of optimisations, and is necessary to avoid the prohibitively large number of tests if the space was explored exhaustively. FFD allows a previously intractable number of optimisations to be explored at the same time, and their interactions accounted for.

An example of a *full* factorial design (exhaustive) in shown in Figure 4.5a. This is simply every combination of the optimisations X_1, X_2 and X_3, resulting in 8 tests. The number of tests scales exponentially in a full factorial design:

$$T = 2^N, \tag{4.1}$$

where N is the number of optimisations and T is the number of tests that need to be run. The effect a single optimisation has on the performance or energy can be estimated by taking the difference between the average performance when the optimisation is on, and the average performance when the optimisation is off.

$$K_o = \frac{\sum S_{on}(o)}{|S_{on}(o)|} - \frac{\sum S_{off}(o)}{|S_{off}(o)|}. \tag{4.2}$$

In the above equation, S_{on} and S_{off} return the set of optimisations which are enabled and disabled respectively. The parameter, o, indicates a specific optimisation and K_o gives the main effect of this optimisation. This equation is illustrated in Figure 4.6. A similar technique can be used to estimate the effect of second and higher-order interactions. In the above equation, o is replaced by a set of optimisations.

A full factorial experiment design, however, captures many interactions which may not be relevant — higher-order interactions are statistically unlikely [99]. In this case, a *fractional* factorial design can be used instead. A fractional factorial design reduces the number of tests

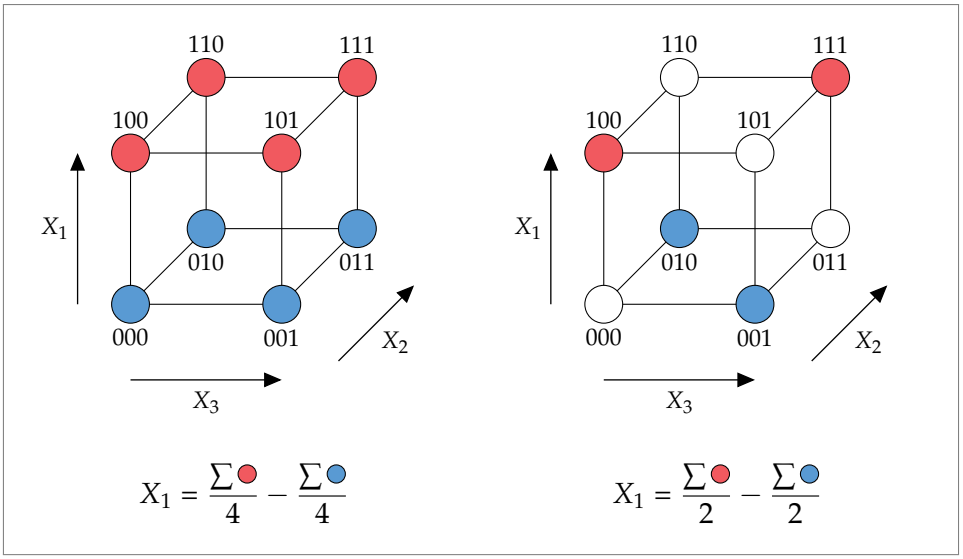


Figure 4.6: The method of calculating the effect due to an optimisation. In the figure, the effect of the optimisation X_1 is calculated.

required in an experiment to an arbitrarily low number. The number of tests required is limited by the desired number of aliasing factors. This defines the 'resolution' of the design. In an experiment which expects to have few higher-order interactions, a low-resolution design can be used and the space can be explored with very few tests. As compiler optimisations are known to have many interactions [100], a higher-resolution design is needed.

Figure 4.5b shows a 'half-fraction' design, derived from the design in Figure 4.5a. This design reduces the number of runs from 8 to 4. By halving the number of runs, the tests can be completed quicker, albeit with some information loss about the interactions between factors. The exact information lost is given by the *aliasing structure* and the *generator polynomial(s)* of the fractional factorial design [85]. The generator polynomial for the fractional factorial design in Figure 4.5b is $I = X_1X_2X_3$. The values of the parameters for each run can be generated from this polynomial by assigning the values -1 (off) or 1 (on) to the factor and ensuring the equation results in $I = 1$. The aliasing structure of a design specifies how many interactions of the factors can be discerned from each other.

Another fractional factorial design with equivalent aliasing structure can be generating by ensuring all values result in $I = -1$. Both these sets of runs are shown in Table 4.1, and are equivalent in the amount of information that can be gained about a factor from them.

The generator polynomial describes the aliasing structure of the design. For example, in the above fractional factorial design, the generator can be rearranged into three possible equations:

$\mathbf{X_1 = X_2X_3}$ Factor X_1 is aliased with the two-way interaction between X_2 and X_3.
$\mathbf{X_2 = X_1X_3}$ Factor X_2 is aliased with the two-way interaction between X_1 and X_3.
$\mathbf{X_3 = X_1X_2}$ Factor X_3 is aliased with the two-way interaction between X_1 and X_2.

This aliasing structure means that every one-factor main effect is confounded with another two-way interaction, and only gives a good estimate of the main effect if the two-way interactions are insignificant. This is known as a resolution III design. Resolution V designs are more typically used, since the main effect is only aliased with fourth order factors — this allows a good estimate of the main effect when two- and three-way interactions may be significant. For example, a

Run	X_1	X_2	X_3
1	1	1	1
2	1	−1	−1
3	−1	−1	1
4	−1	1	−1

(a) Runs for $I = 1$.

Run	X_1	X_2	X_3
1	−1	1	1
2	−1	−1	−1
3	1	−1	−1
4	1	−1	1

(b) Runs for $I = -1$.

Table 4.1: Fractional factorial designs for the generator $I = X_1X_2X_3$.

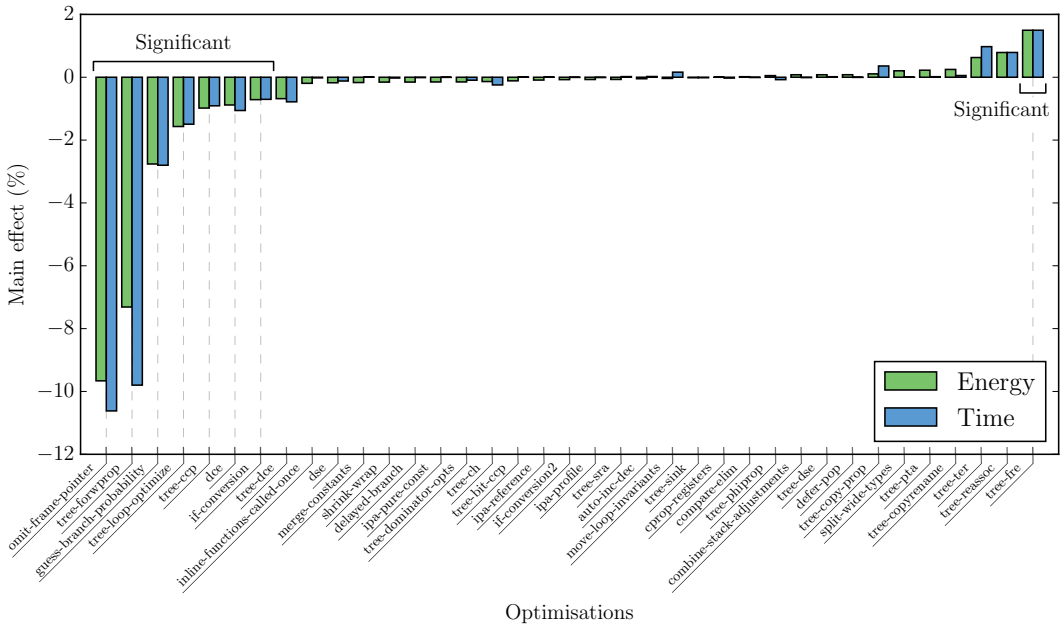


Figure 4.7: The blowfish benchmark run on the STM32F0 SoC (Cortex-M0, 01).

resolution V design means that the effect of a single optimisation cannot be discerned from the additional effect of a combination of four optimisations being enabled simultaneously.

The reduction in number of runs (tests) allows a large combinatorial design to be explored in a reasonable time-frame: a set of 36 optimisations would be 68 billion tests if every combination was tested, however can be sufficiently covered with a fractional factorial design (resolution V) with 2048 runs. The statistical significance of the result can be tested using the Mann-Whitney U test [84, 1]. This statistical test measures the likelihood of one distribution typically having a larger value than the other, allowing the optimisations that have a significant effect to be determined.

4.4.2. *Individual optimisation analysis*

Fractional Factorial Design (FFD) can be used to analyse individual optimisations for their efficacy at reducing energy consumption. A FFD was run for each of the benchmarks in the suite (see Chapter 3) on a total of five platforms.

Figure 4.7 shows the results of one FFD, the blowfish benchmark run on a STM32F0 SoC. The

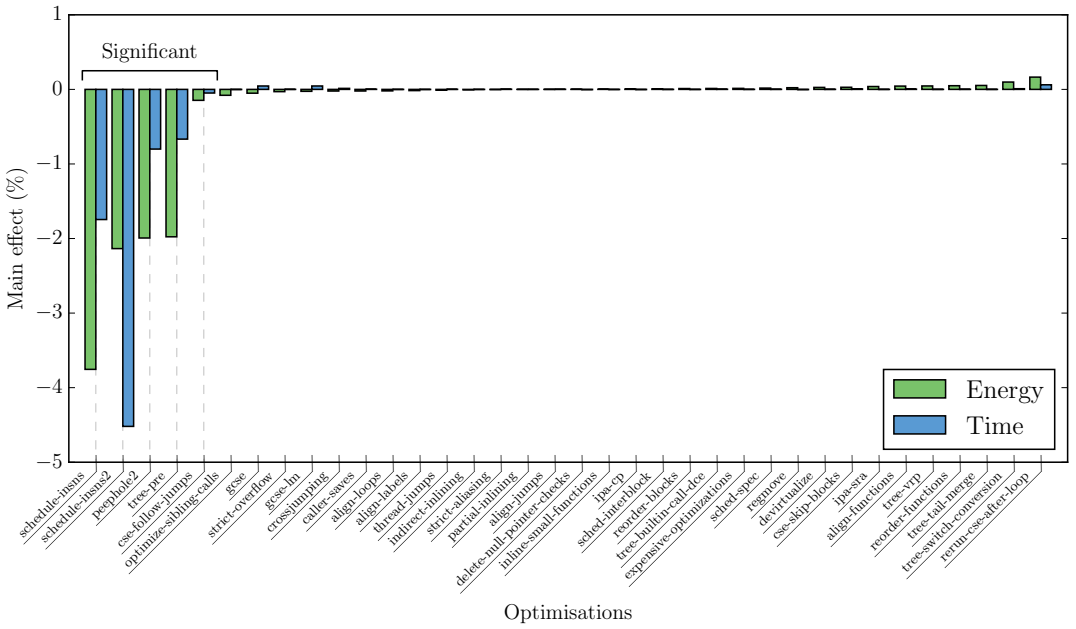


Figure 4.8: The fdct benchmark run on the STM32F1 SoC (Cortex-M3, `O2`).

graph shows individual optimisations from GCC 4.7's `O1` optimisation level. The main effect (the effect attributable to a single optimisation, rather than a combination of optimisations) for each optimisation is plotted, for both time and energy, and the significant results as determined by the Mann-Whitney U test are indicated by the bracket above or below the optimisations. The most effective optimisation for this benchmark is *omit-frame-pointer*, an optimisation which frees an additional register for use in general purpose calculations (see page 105 for more details). This has the effect of significantly decreasing both energy and execution time on average (by similar amounts). Most optimisations in this graph reduce the energy consumption of the processor purely due to reducing the total run-time of the benchmark — the average power during the benchmark is constant. The low complexity of the pipeline in the Cortex-M0 means there is not a large amount of scope for divergence between energy and time. Many optimisations in the middle of the graph are shown to have a low or insignificant effect, with the majority of these fluctuations being influenced by measurement noise.

A similar trend is seen in Figure 4.8, on the more complex STM32F1 SoC. As in the STM32F1, an optimisation that improves energy consumption also improves execution time. However, now the effect on energy and time is not always proportional. The two most effective optimisations both perform instruction scheduling, with `schedule-insns` performing instruction scheduling before register allocation and `schedule-insns2` performing it after (see page 102). For scheduling after register allocation, the greater reduction in time can be explained by the presence of load/store pipelining in the Cortex-M3. Register allocation may insert additional spill instructions, which the schedule phase attempts to place adjacent to one another, so that the address and data phases of the memory access can be pipelined [101]. The total energy is only lowered slightly, because only the base power overhead of the additional cycles is removed.

On the other hand, scheduling before the register allocation has the opposite effect — energy consumption is reduced by more than execution time. Scheduling before register allocations

Benchmark	STM32F0			STM32F1			AM335x			Epiphany			XMEGA		
	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd
2dfir	W	**A**	.	**A**	I	G	M	**A**	H	I	**B**	D	T	**A**	Z
blowfish	K	J	H	K	J	H	K	J	R	D	X	I	K	J	R
crc32	F	E	**A**	F	E	**A**	F	E	G	.	.	.	F	.	.
cubic	**B**	**C**	.	**B**	**C**	.	**B**	**C**	.	**B**	**C**	Y	**B**	**C**	.
dijkstra	F	**C**	**B**	E	**C**	**B**	E	**C**	**B**	**C**	**B**	.	F	**C**	**B**
fdct	L	**A**	E	L	V	**B**	Q	G	V	**B**	I	D	G	**B**	**C**
matmult-int	**A**	O	P	**A**	L	P	**A**	H	P	**B**	I	D	U	**A**	W
matmult-float	**A**	.	.	**A**	L	F	H	**A**	**B**	D	I	**B**	**A**	T	G
rijndael	H	D	N	N	**A**	α	S	Q	N	D	S	**A**	G	O	N
sha	**A**	M	G	**A**	M	E	**A**	M	Q	D	**A**	β	**A**	O	U

Table 4.2: The most effective optimisations for each benchmark and platform. The key for each letter is given in Table 4.3.

greatly affects the amount of spill code that is generated and if dependent operations are clustered together, the register allocator may be able to minimise the total spill code. Minimising the spill code has the overall effect of minimising energy more than time, since memory access have an above average power dissipation — removing them will lower the average power, and therefore energy.

4.4.3. *Optimisation combination analysis*

Individually, optimisations can have a significant effect on reducing energy (according to the Mann-Whitney U test) on a specific benchmark for a specific target, however, there is not a single optimisation or set of optimisations which perform well for all benchmarks and targets. The set of effective optimisations for each benchmark and platform combination can be found by extracting the top most results from each of the fractional factorial designs.

Table 4.2 shows the top three optimisations for each combination of benchmark and platform, extracted from the results of all optimisations enabled by `O1`, `O2` and `O3`. The highlighted cells represent the most frequent effective optimisations, and the empty cells occur when no significantly effective optimisation was found. This table does not exactly correspond to the earlier graphs (Figures 4.7 and 4.8), since the table lists all optimisations rather than just plotting the optimisations within a single optimisations group. For example, while `schedule-insns` is the most effective optimisation in Figure 4.8 (*blowfish* on STM32F1, at `O2`), it is overall less effective than the optimisations **K**, **J** and **H**, which had effects of -22%, -17%, and -7% respectively (and are all from the `O1` group).

The most common optimisation is `tree-loop-optimize` (**A**). The optimisation performs a set of simple loop optimisations such as loop invariant motion and loop unswitching (see Appendix A). Since the optimisations focus on increasing loop performance, the benchmarks that are particularly loop intensive are affected the most — *matmult-int*, *matmult-float*, *rijndael* and *sha*. The second most common optimisation is `tree-dominator-opts` (**B**), performing a collection of simple optimisations during the dominator tree traversal. These optimisations include constant propagation, copy propagation, redundancy elimination, range propagation, expression simplification and jump threading (see pages 99–106). This optimisation ranks highly on all platforms for the *cubic* and *dijkstra* benchmarks, as well as being effective in general for the Epiphany processor. However, this optimisation is not as effective for any other benchmark or platform, suggesting that the structure of these two benchmarks is receptive to these types of transformation.

ID	Count E	Count T	Flag	ID	Count E	Count T	Flag
A	22	22	`tree-loop-optimize`	O	3	3	`guess-branch-probability`
B	17	16	`tree-dominator-opts`	P	3	4	`inline-small-functions`
C	11	11	`tree-fre`	Q	3	5	`schedule-insns2`
D	8	8	`dce`	R	2	2	`tree-forwprop`
E	7	7	`move-loop-invariants`	S	2	3	`schedule-insns`
F	7	9	`inline-functions`	T	2	2	`gcse`
G	7	7	`tree-ter`	U	2	2	`regmove`
H	6	4	`omit-frame-pointer`	V	2	2	`ira-loop-pressure`
I	6	7	`tree-ch`	W	2	2	`tree-pre`
J	4	4	`inline-functions-called-once`	X	1	1	`ipa-profile`
K	4	4	`ipa-pure-const`	Y	1	1	`combine-stack-adjustments`
L	4	2	`ipa-cp-clone`	Z	1	0	`auto-inc-dec`
M	4	3	`predictive-commoning`	α	1	1	`tree-pta`
N	4	4	`tree-sra`	β	1	0	`dse`

Table 4.3: The optimisation flag corresponding to each letter in Table 4.2. The E column is the frequency of effective optimisations for energy, and the T column is an equivalent count but for execution time (full table equivalent of Table 4.2 not shown). While 82 optimisations were present in the full set, many were never effective enough to be in the top three.

A similar pattern is seen with the third most-effective optimisation (`tree-fre`) — there are specific benchmarks for which the optimisation is particularly effective across many of the platforms. Logically, benchmark structure is one of the most important things in determining whether or not the optimisation will be effective, because the optimisation attempts to transform a specific pattern in the code.

There is a set of optimisations which is often effective on the Epiphany processor (**B**, **D** and I), whereas there is no set of optimisations which are effective for the other processors. This is likely due to parts of the compiler being significantly different from the other platforms (which share certain parts of the compiler, since the architectures are all related). In addition to the `tree-dominator-opts` optimisation flag explained above, dead code elimination (**D**) and loop header copying (**I**) are the frequently effective optimisations for the Epiphany (see Appendix A, pages 100 and 103). Dead code elimination can be divided into two types — unreachable code elimination and unused code elimination. In this case, removing unused code can speed-up execution since there is less total work to perform, however removing unreachable code can also decrease execution time. In combination with other optimisations, removing unreachable code can remove branches, and increase the efficacy of the static analysis (through less control flow), which enables other optimisations.

There are commonalities seen between benchmarks on the ARM based platforms (STM32F0, STM32F1 and AM335x), however, these are not due to their instruction set, since each of these processors uses a different instruction set — Thumb, ThumbV2 and ARM mode respectively. However, the total number of addressable registers is the same for each instruction set, suggesting that the effectiveness of some of the optimisations is limited by the register pressure on these platforms. The `omit-frame-pointer` optimisation is effective because of the high register pressure — this optimisation allows an extra register to be used for general purpose computation.

Table 4.4 lists the optimisations which *increase* the energy consumption of that benchmark on the specified platform by the largest amount. There are few common optimisations across a single benchmark or platform in this table, because of the general design of the optimisations. A typical optimisation is designed to recognise a specific pattern in the source code, and transform

Benchmark	STM32F0 1st	2nd	3rd	STM32F1 1st	2nd	3rd	AM335x 1st	2nd	3rd	Epiphany 1st	2nd	3rd	XMEGA 1st	2nd	3rd
2dfir	.	.	.	.	.	.	O	.	.	.	.	.	**B**	I	.
blowfish	S	.	.	D	.	.	**C**	.	.	N	.	.	.	.	.
crc32	.	.	.	.	.	.	.	.	.	Q	.	.	.	.	.
cubic	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
dijkstra	.	.	.	**A**	.	.	**A**	D	.	.	.	.	.	.	.
fdct	D	.	.	G	**A**	.	G	**A**	.	E	.	.	**B**	G	.
matmult-int	E	.	.	.	.	.	.	.	.	.	.	.	**B**	.	.
matmult-float	.	.	.	.	.	.	.	.	.	H	J	.	**B**	R	.
rijndael	F	M	.	F	**A**	.	E	.	.	.	.	.	T	U	.
sha	F	P	.	L	K	**C**	**C**	K	L	J	H	.	I	.	.

Table 4.4: The least effective optimisations for each benchmark and platform. The key for each letter is given in Table 4.5.

ID	Count	Flag	ID	Count	Flag
A	5	`schedule-insns`	L	2	`tree-dominator-opts`
B	4	`omit-frame-pointer`	M	1	`tree-forwprop`
C	3	`tree-fre`	N	1	`tree-copyrename`
D	3	`ira-loop-pressure`	O	1	`inline-small-functions`
E	3	`tree-ter`	P	1	`move-loop-invariants`
F	3	`tree-reassoc`	Q	1	`tree-copy-prop`
G	3	`tree-loop-optimize`	R	1	`tree-pre`
H	2	`if-conversion2`	S	1	`tree-ch`
I	2	`predictive-commoning`	T	1	`tree-vrp`
J	2	`cprop-registers`	U	1	`regmove`
K	2	`gcse`			

Table 4.5: The optimisation flag corresponding to each letter in Table 4.4 and frequency of the optimisation flag in the table.

it into a pattern which is believed to perform better. Occasionally, for a specific combination of benchmark and optimisation a piece of code is transformed and leads to poorly performing output code, but this is not as common.

The only optimisation which is frequently negative for a particular SoC is `omit-frame-pointer`, on XMEGA, having a negative effect on four of the 10 benchmarks. It is expected that this optimisation would have only marginal effect on this architecture, because the AVR has 32 registers and typically low register pressure. However, on examination of the generated code, the compiler inserts a larger number of load and store instructions to stack-based variables when enabling this optimisation, leading to an increase in execution time.

The `schedule-insns` optimisation appears most frequently in the table, clustered around *dijkstra* and *fdct* on the STM32F1 and AM335x. Since this optimisation schedules instructions before register allocation, the scheduling causes a suboptimal register assignment for these particular benchmarks, increasing spill code and therefore energy consumption.

The third most common negative optimisation is `tree-fre` — 'full redundancy elimination'. The optimisation considers the case where there are multiple identical expressions on all control paths for a region in the program (see page 106). The optimisation increases energy consumption for the AM335x on two benchmarks, whereas it decreases energy on the same platform for two different benchmarks. In this case the optimisation is likely inhibiting a subsequent optimisation

which would reduce the energy consumption, since generally removing redundant computation should increase performance and decrease energy.

There are many fewer optimisations seen to increase the energy consumption, with some benchmarks having no optimisations which have a significant effect. This is due to the decision of whether an optimisation can be applied being decided based on heuristics from the source code and the heuristic being correct in the majority of instances.

This analysis showing the top optimisations for energy ignores the time component which may be partially responsible for the reduction in energy consumption. Table 4.3 also contains the most frequently effective optimisations when performing the same analysis for execution time, instead of energy. The most commonly effective flags for execution time are very similar to that of energy, adding weight to the argument that it is really the execution-time reduction which is leading to the energy efficiency.

Overall the selection of the best optimisation is highly dependent on the benchmark, and on the compilation target, motivating the need for a better strategy for selecting optimisations. Many optimisations are effective across platforms for specific benchmarks, however there are also optimisations whose effectiveness depends on the processor target. The most common effective optimisations are those which have grouped together smaller optimisations but without having the ability to separate individual passes, rather than the grouped optimisation flag it is impossible to tell whether all the constituent optimisations are effective. However, all the individual passes implement transformations which reduce the total amount of work performed in the applications and so are likely to be beneficial in most cases.

The best and worst optimisations for both energy and time are very similar – the majority of energy reduction is due to an execution time saving. This is expected, since most of the optimisations are attempting to reduce the total amount of code executed, or reorganise the code such that it executes faster.

4.5. Choosing optimisations using genetic algorithms

The analysis in the previous section determined which optimisations have the most impact on the benchmarks, but since it is a statistical analysis it does not necessarily imply that best performing optimisations form the best performing set to pick. It is possible that an individual optimisation may overall be ineffective, but has a large effect on the energy or time when other optimisations have structured the code in a specific way. In this section, a genetic algorithm is used as a way towards finding the best possible configuration of optimisations. By attempting to optimise the energy consumption, rather than just analyse which optimisations increase or decrease the energy, the hypothesis of existing compiler optimisations improving energy via a reduction in execution time can be explored.

4.5.1. *Genetic algorithms*

Genetic algorithms are a biologically-inspired method for choosing parameters to an optimisation problem. A genetic algorithm is used here to choose the optimisations to apply to the source code, allowing unexpected combinations of optimisations to be explored. The genetic algorithm evaluates each possible solution using a fitness function, which can be tuned to different goals. By giving different goals of minimising energy consumption or execution time, the genetic algorithm will produce different solutions.

The genetic algorithm generates a pool of random *individuals*, where each individual (here, a bit string) specifies which optimisations are enabled, as in the left side of Figure 4.9. Each bit in the string represents an enabled or disabled optimisation. Each individual in the pool has

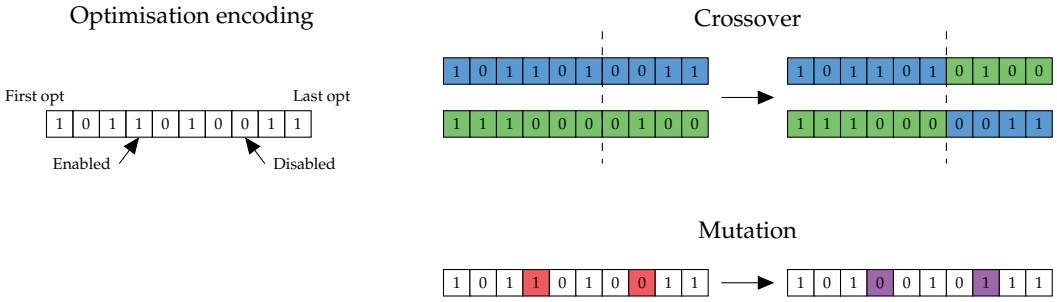


Figure 4.9: The encoding used for a standard genetic algorithm and illustration of the crossover and mutation operators.

its energy and execution time measured, then the fitness function applied (see below). Each individual in the population can then be ranked by the fitness function, and the top n selected to be propagated to the next generation. Each optimisation string that is selected is first crossed-over with another, then mutated, with a random number of mutations (see the right side of Figure 4.9).

As the number of generations increases, the optimisation strings with the best fitness are kept and combined together. This eventually leads to a set of optimisation strings which have a high fitness — whether that is a low energy consumption or a low execution time, as selected by the fitness function.

The genetic algorithm was applied to each of the benchmarks, with a variety of fitness functions. A pool size of 30 optimisation strings was selected with a mutation rate of 5%. The mutation rate specifies the likelihood of each bit in the individual being inverted, and was increased for each generation which did not see an improvement on the previous. The genetic algorithm ran for 100 iterations, and the best set of optimisations for each benchmark and energy/time was recorded.

4.5.2. *Fitness functions*

A variety of fitness functions are used, allowing some of the main hypotheses to be tested. The fitness functions are all of the form $F(e, t)$, where e is the energy consumption of the test, and t is the execution time of the test. The fitness function returns a value, where a higher value represents an individual which better fits the optimisation goal.

Minimise energy. $F(e, t) = \frac{1}{e}$**.** This is the goal of minimising the total energy consumption of the benchmark, without considering execution time. The goal is likely to reduce execution time as well, if the hypothesis of energy consumption and execution time being almost proportional is true.

Minimise time. $F(e, t) = \frac{1}{t}$**.** This is the goal of minimising the total execution time of the benchmark, without considering energy consumption, but is likely to reduce energy consumption too. A useful comparison can be made with the previous result — a similar result should be obtained if energy consumption and execution time are similar the majority of time.

Minimise power. $F(e, t) = \frac{t}{e}$**.** To minimise power, both energy consumption and execution time are used. Since there are no additional constraints on energy or time, it is possible the goal is just as likely to select a result which decreases power by increasing just the execution time, as it is to select a result which reduces the power by reducing energy consumption.

However, this metric indicates an approximate lower bound on the power dissipation of the benchmark.

Maximise power. $F(e,t) = \frac{e}{t}$**.** The goal of maximising power dissipation attempts to find an optimisation sequence which causes the highest average power dissipation. The goal is used to find an approximate upper bound on the power dissipation. With the results of the minimise power dissipation goal, the range between them should show how much divergence between energy and time a set of compiler optimisations can achieve.

4.5.3. *Results*

For each benchmark and fitness function, the genetic algorithm was run on the STM32F1 processor. This processor was chosen because it is representative of deeply embedded processors and is one of the most commonly used of the available SoCs. The results are shown in Figure 4.10, clustered by benchmark, then by fitness metric. The energy consumption (■) and execution time (■) are shown for the first two fitness goals, and the average power (■) is shown for the minimise- and maximise-power goals.

In all cases, the goals targeting lower energy consumption and lower execution time produce a set of optimisations which perform similarly. This adds weight to the hypothesis that with the existing compiler optimisations, optimising for time is the same as optimising for energy. The genetic algorithm is able to significantly lower the energy and time, by up to 30% further over the `O3` optimisation sequence.

The goals attempting to maximise or minimise power show more range, but are still within $\pm$14% of the baseline power for all benchmarks. Since this is an approximation of the minimum and maximum power the benchmark can execute at, it puts limits on how much the current optimisations in the compiler affect the power dissipation, rather than achieving energy efficiency through speed. The benchmark that has the lowest average power is *matmult-int*, at 14% lower power dissipation, suggesting that the optimisations cannot significantly reduce the average power as much as the execution time is lowered (22% for *matmult-int*).

The results for power also cause the energy and time metrics to vary wildly, in some cases increasing the execution time of a benchmark by almost 50% to achieve a 10% lower power (*dijkstra*). The single focus of the metric on power is the cause of this, and another fitness term would have to be added to the goal to prevent execution time or energy from increasing too much for practical use.

4.6. Conclusion

The 82 currently available optimisations existing in the compiler (GCC) have been extensively evaluated, from the optimisation level grouping (`O0`–`O3`), to individual optimisations. In almost all cases, there is a close correlation between execution time and energy consumption. The overall optimisation levels tend to make the application faster and more energy efficient in proportion, with a few differences in the more complex processors. For the existing optimisations, the complexity of the processor's pipeline affects how much divergence is possible between energy consumption and time.

Each of the optimisation flags were explored individually, using fractional factorial design to minimise the total number of tests necessary, while still accounting for interactions between optimisations. With this methodology some optimisations were found to affect the proportionality between energy and time, such as instruction scheduling, and optimisations which change the

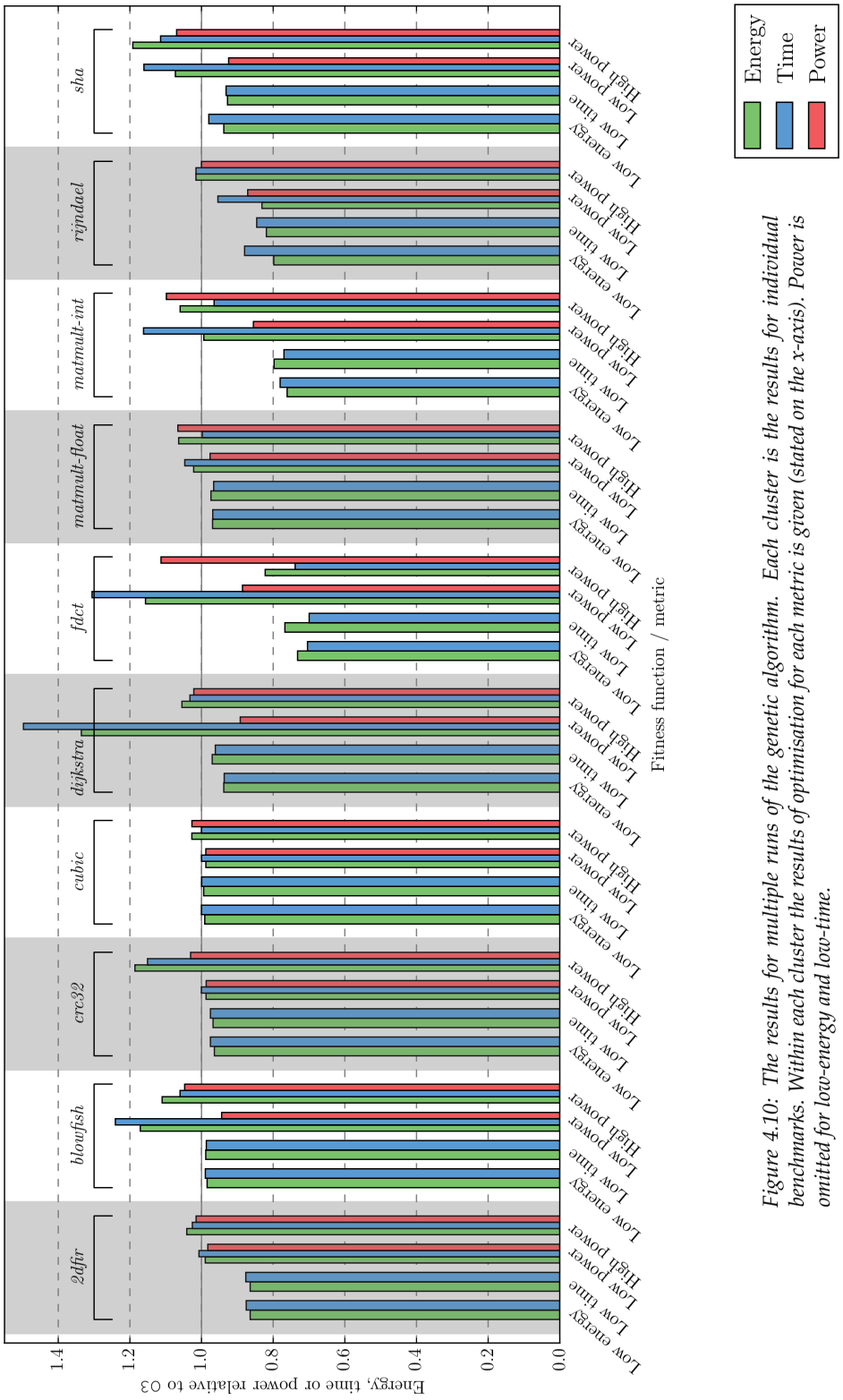


Figure 4.10: The results for multiple runs of the genetic algorithm. Each cluster is the results for individual benchmarks. Within each cluster the results of optimisation for each metric is given (stated on the x-axis). Power is omitted for low-energy and low-time.

number of memory accesses. Apart from these optimisations, most others purely reduce the execution time, and the energy efficiency follows from that reduction.

Many optimisations are effective for multiple benchmarks, but there are a few which are more widely effective than others. Most of these optimisations are collections of simple transformations, such as common subexpression elimination and constant propagation. Other optimisations are particularly profitable for certain benchmarks — the exact structure of the program lends itself to a specific type of transformation. There are also some optimisations which are seen on a per-platform basis. There are many optimisations which never appear to have a large effect (i.e. appear in the top three). Some of these optimisations are effective, but have a smaller effect, and some do not apply to either the platform or the benchmark. There are fewer commonalities amongst the optimisations which perform poorly and increase the energy consumption, but there are individual optimisations which exhibit negative effects for a specific combination of benchmark and platform — these are likely because the compiler's heuristics wrongly indicated it should apply the transformation. For example, the `omit-frame-pointer` optimisation performs poorly on the XMEGA platform — triggering additional loads and stores to be inserted by the register allocator.

An approximation of the best possible optimisation selection is found using a genetic algorithm, with various goal functions. When using energy or time as the goal metric, the search produces very similar results — the search is not able to find a set of optimisations which reduce the energy consumption at the expense of execution time and vice versa. A similar exploration was undertaken by setting the goal of minimising power as much as possible. The genetic algorithm was only able to find sequences which reduce power by up to 14%, with an average of 7%. This suggests that existing optimisations are not able to reduce energy significantly without also affecting the execution time — there is a limit to the amount of average power change that existing optimisations can cause.

This chapter answers the first set of research questions: "Do existing compiler optimisations save energy purely by reducing the k_T coefficient?", where k_T is the effect of the optimisation on the execution time, and therefore energy (see Chapter 2). The majority of the optimisations reduce the execution time (and by definition k_T), and this affects the energy consumption of the program being tested. There are very few optimisations for which k_P, the optimisation's effect on power, is significantly different from 1. Instruction scheduling can increase k_P (see Figure 4.8 on page 35), while reducing memory accesses decreases k_P. Overall, these optimisations are sometimes able to save energy, when the execution time is decreased enough, i.e. $k_T < \frac{1}{k_P}$.

Overall for embedded platforms and existing compilers, compiling for energy is very similar to compiling for execution time, since most of the optimisations change the program in a way that either makes the program faster with the same average power, or reduces the total amount of work, also reducing energy consumption in proportion to execution time. This motivates the search for a new class of optimisations which attempts to reduce energy consumption by reducing the average running power of the SoC. The rest of this thesis investigates this class of optimisations.

This page is intentionally blank.

Chapter 5.
Optimisations designed for energy consumption

Work in this chapter also appears in the following publications:

- James Pallister, Kerstin Eder, Simon J. Hollis and Jeremy Bennett. "A high-level model of embedded flash energy consumption". In: *CASES14 Proceedings of the 2014 international conference on Compilers, Architecture, and Synthesis for Embedded Systems.* New Delhi, India. ACM Press, 2014, p. 74.
- James Pallister, Kerstin Eder and Simon J. Hollis. "Optimizing the flash-RAM energy trade-off in deeply embedded systems". In: *CGO'15 Proceedings of the 2015 international symposium on Code Generation and Optimization.* San Francisco, USA. ACM Press, 2015.

5.1. Introduction

Historically, optimisations have been added into compilers to increase the speed or decrease the code size of an application. Optimising for these metrics also affects energy consumption. For example, increasing the speed of a program will decrease the total energy consumption due to less work performed, or less leakage (since the processor can be put into a low power mode sooner), and decreasing the code size may reduce energy consumed inside a cache or prefetch buffer since fewer costly memory accesses need to be performed.

As seen in the previous chapter, most of the existing optimisations change energy and execution time in proportion, with few optimisations causing a large deviation from this correlation. This suggests there may be other possible optimisations which specifically target energy consumption by reducing the average power, rather than purely reducing execution time. Some optimisations have been proposed by previous work, and this chapter identifies two energy characteristics which are then developed into optimisations that fall into this new class. These optimisations reduce energy by attempting to lower the average running power of the software running.

Optimisations for energy consumption are typically much more target specific than optimisations for performance or code size — e.g. redundancy elimination (see page 106) will almost always reduce code size to some degree, for all benchmarks and all SoCs. In contrast to performance optimisations, energy optimisations often exploit processor or chip-specific functionality and characteristics, making them more challenging to develop.

This chapter discusses two novel energy optimisations, which can be achieved by using information about the target processor. The optimisations are different from performance optimisations in that they affect the positioning of the code that is executed, rather than radically transforming the code.

Code alignment. It is observed that the alignment of code during execution from embedded flash has an impact on the energy consumption. In embedded systems, code is often executed directly from flash, and when these accesses cross particular internal boundaries a corresponding additional energy cost is incurred. Since the code's execution time does not change with its alignment, this enables an optimisation which lowers average power.

The phenomenon is modelled (Section 5.3) and then an optimisation using this model is

developed (Section 5.3.2). The optimisation is not able to save significant energy due to the precise characteristics of the SoCs tested.

RAM overlay. In some processors code can be executed from either flash or RAM, with little difference in execution time. However, the execution from flash is significantly more power hungry than from the RAM. Thus, if all the code were to be executed from RAM, significant energy savings could be achieved. Unfortunately there is typically 8–16 times more flash than RAM, meaning only some of the code can be moved to RAM. Other complications include difficulties in jumping between the two memory spaces, and having both fetches and data accesses to the RAM can cause contention on the memory buses, resulting in higher latency.

These trade-offs are explored in Section 5.4, and the optimal solution found using Integer Linear Programming (ILP). This is used to implement the RAM overlay optimisation, saving an average of 10% and up to 26% of the SoC's energy.

This chapter first discusses existing attempts at creating optimisations for energy consumption. Then, the analysis and optimisation based on code alignment in flash memory is presented and evaluated. Then, the RAM overlay optimisation is presented and evaluated. Finally, both optimisations are discussed along with their applicability and efficacy.

5.2. Background

Many studies develop new optimisations to target energy consumption or power dissipation. Roughly these attempts fall into three categories: attempting to minimise the amount of bit flipping inside the processor, optimising resource usage (including scheduling), or disabling parts of the processor to minimise the leakage power dissipation.

Various studies examine the impact of existing optimisations on energy or power, and tune the optimisations for these metrics. However, optimisations specifically targeting energy consumption can be developed. These optimisations may increase performance (although not as their primary goal), but often present a trade-off between execution time and energy consumption. This section primarily looks at optimisations which attempt to minimise energy consumption without architectural modifications — these are optimisations which can be implemented with existing processors.

Dynamic voltage and frequency scaling. DVFS exploits the non-linear scaling between voltage and power consumption, and the relationship between voltage and frequency. The energy consumed by a single bit flip in a transistor is lower if the voltage is reduced, however, it requires a larger amount of time to perform the transition. Since the energy for each bit flip is proportional to the square of the voltage, decreasing the frequency and decreasing voltage results in an energy saving, but places limits on the maximum clock frequency for a given voltage [102].

Hsu et al. [103] develop an algorithm to decide when scaling down the CPU's frequency and voltage is beneficial. The scaling is applied where the CPU can be slowed without significant loss in performance achieving up to 28% reduction in energy consumption at a 5% increase in execution time. The algorithm uses profile information about each region of the program along with predetermined voltage-frequency pairs to determine when and where to change the voltage frequency of the CPU.

The same approach is taken in [104], applied to an embedded processor. This processor only had two possible voltage-frequencies pairs and a lower level of energy savings was

found with this configuration.

Register file and allocation optimisations. Several studies have attempted to optimise register allocation, since memory operations are typically more expensive in time and energy than data manipulation instructions. Memory accesses are also proportionally more expensive in energy consumption, since they traverse a large portion of the system's circuitry and area.

Zhang et al. [105] attempt to re-allocate registers as a post-compilation pass, finding that the number of memory loads and stores can be reduced. The reduction is analysed with the cache simulator, CACTI [31], and an architectural-level power analyser, WATTCH [30], and found to reduce data cache energy consumption by up to 15% and total cycle count by 34%, on a Pentium processor, thus indicating that much of the energy saving was actually due to a lower execution time.

Another study attempted to move global variables into registers to reduce cache power dissipation [106]. Assigning global variables with a long lifetime to registers reduces the overall number of loads and stores, potentially increasing efficiency more than variables with smaller live ranges which do not completely utilise a register. Additionally, global variables are often non-local to the program's current working set, and this may stop necessary data from being evicted from the cache, preventing future cache misses. The technique was effective at reducing energy in some cases, however, it was hard to predict the optimisation's effect due to its complex relationship with register pressure.

Some studies have attempted to minimise the energy cost due to register selection [107, 108]. Registers are renamed so that consecutively accessed registers have similar register numbers, with few bit flips. These studies achieved large reductions in the amount of bit flips between register numbers. However, the register selection bus is typically a small part of the processor, so these optimisations may only marginally reduce energy.

Instruction scheduling. Instruction scheduling has been explored as a means to minimise bit-flips between consecutive instructions. The bus between main memory and the processor or caches is often power hungry to toggle (long wires resulting in higher capacitance and higher energy to flip) and is a target for minimising flips. Scheduling the instructions of a program such that they cause fewer bits to toggle on the bus should reduce the overall energy consumption. Tomiyama et al. [109] achieve 28% fewer transitions in the instruction stream but do not translate this to energy savings.

Other work by Parikh et al. [2] explores different scheduling constraints and applies an energy model to the generated code, resulting in up to a 10% decrease in energy consumption. However, this relies on significant circuit switching effects (up to 150% of the base instruction energy cost in the energy model) which is highly dependent on processor architecture and shown to be less than 10% in other processors [13].

Toburen et al. [110] apply a similar optimisation to VLIW (Very Long Instruction Word) processors — scheduling instructions simultaneously up to a maximum energy limit per cycle. This was achieved by assigning an energy cost to each functional unit and iteratively scheduling instructions from a dataflow DAG (Directed Acyclic Graph) until the energy limit was reached and resulted in a flatter power profile for the application.

Instruction selection. The actual selection of instructions can be altered based on their energy consumption. Strength reduction is an optimisation which exchanges two mathematically identical instructions based on some metric. Often this optimisation is used to remove multiplies from code, for example `r0` $\times$ 2 can be changed for `r0` + `r0` in many processors

where addition is faster than multiplication. This technique can also be used to improve energy [24]. In many embedded processors both multiplication and addition can take the same number of cycles. However, a multiply instruction is often more power hungry, due to the comparatively large amount of circuitry to implement it.

Instructions can also be selected based on multiple criteria. Wu et al. [3] examine the case where two instruction sets are present in the processors — a 32-bit, powerful instruction set (all registers and functions possible), and a more restricted 16-bit instruction set, in this case ARM and Thumb respectively. The study uses a multi-objective ant colony algorithm to select which functions should be encoded using the smaller instruction set, showing that a good trade-off between execution time and code size can be achieved. It is likely that a similar technique would allow selection based on energy.

Inserting sleep modes. Sleep modes are often an effective way of reducing static power since the majority of the processor can be turned off, with only certain areas of the chip remaining powered.

Min et al. [111] describe a framework for deciding when a sleep state should be entered based on the expected rewards, including the prediction of interrupts. This resulted in saving significant energy while only minimally impacting performance.

Resource scheduling is another technique which can enable sleep modes to be utilised effectively. By scheduling actions such that they coincide with each other, the necessary components or processor can be enabled fewer times, as explored by Venkatachalam et al. [112]. Scheduling reduces the number of transitions between sleep modes and running modes and thus energy. This is very similar to scheduling tasks such that they take a minimal amount of energy, as in Yao et al. [18].

Scratchpad memory utilisation. Scratchpad memories are implemented in many processors as an area of on-chip, fast RAM. The speed of this RAM results in large energy savings, partly due to the performance increase and partly due to the reduction in expensive off-chip accesses. This memory can be managed by the compiler, which automatically places code and data into the scratchpad memory [113].

A technique for moving code and data objects to a scratchpad memory was compared to a similarly sized cache in Steinke et al. [114], finding the scratchpad memory almost always significantly outperformed the cache for both execution time and energy consumption. This is due to the scratchpad memory's lower complexity, and therefore lower energy per access, as well as the advantage of using the compiler's knowledge of the program. The technique was further compared to a static technique also reporting significant energy savings, finding that scratchpad memories use 43% less energy than a cache of the same size (although a significant proportion of this is due to decreased execution time, which was up to 23% lower). The approach formulates a model describing each memory and uses Integer Linear Programming (ILP) to produce a set of objects which should be placed into the scratchpad memory. The assignment was static — code and data were placed in the scratchpad memory at start-up and not moved back to main memory, contrasting with the more flexible cache. An extension to any number of scratchpad memories is produced in [113], where basic blocks can be statically allocated to any memory to minimise execution time and energy consumption. This considers the possibilities that there may be a cost associated with branching between memory spaces.

Kandemir et al. [115] use Presburger formulae (a form of decidable arithmetic without multiplication) to reduce the number of off-chip accesses, storing the data required by array

accesses in a scratchpad memory. The approach is dynamic, selecting a subset of the array and reorganising the access pattern to maximise the use of this subset. Compared to using a cache the approach reduces the off-chip accesses by 39%, however the same optimisation of selecting and reorganising the access pattern is not applied for the cache experiments. A more generic method of dynamically moving code and data into a scratchpad memory is provided by Verma et al. [5], where the problem is shown to be analogous to the global register allocation problem and is solved with ILP.

Further scratchpad memory studies have explored how multiple tasks can cause interference when each task places data into the memory, in Gauthier et al. [116]. A method of minimising the interference is proposed, which attempts to minimise the energy consumption and maximise the number of accesses to the scratchpad memory. An energy reduction of up to 85% was reported, however much of this is likely due to the increase in performance. Kandemir et al. [117] explore embedded multiprocessors each with a scratchpad memory, accessing the same DRAM, and succeed in reducing the energy-delay product by up to 30%.

All of these scratchpad memory studies only consider the case where an alternative memory to the main memory is present, and it is faster or more energy efficient. None consider the case where code is executed directly out of flash and could instead be executed out of RAM for energy savings, as investigated in this thesis. To achieve this, further constraints are needed in the model, to prevent all of the RAM being used, as well as balancing trade-offs between the overhead of being in RAM, the overhead of branching between memories and the energy reduction form executing in RAM. These are discussed further in Section 5.4.

Accuracy reduction. Accuracy of computation can sometimes be reduced without significantly impacting results, to improve both the energy consumption and execution time. EnerJ [118] is an extension to Java adding approximate data types, showing that there were potentially large energy savings if lower accuracy could be exploited.

Eltawil et al. [119] give a survey of many discuss possible trade-offs which may occur between power and reliability, at different levels of the system, from transistors up to software. At the compiler level, various software error correction, and redundancy insertion schemes are discussed. However the majority require hardware support, either to place the processor in a lower accuracy mode, or lowering the voltage to lower power and possibly reduce reliability.

Flash memory. Embedded flash memory is not often studied for its energy impact, with studies focusing on higher-performance devices and modelling SSDs [120]. However, the modelling of NAND flash has been studied at a very fine granularity in [121]. Their tool, FlashPower, uses a large number of parameters describing the flash memory, along with the device feature size, to predict the read, program and erase energies. The tool achieves between 10% and 40% accuracy in a case study and is shown to be useful for design space exploration. In devices with embedded flash, many of the parameters' values are unknown due to manufacturers not disclosing the exact details of their device, making this tool less useful for compiler or developer use.

Code overlays. Moving sections of code between memory spaces has been explored extensively for scratchpad memories. In a system with a scratchpad memory, the main memory is typically slow, and a scratchpad memory can be used as a user-controlled cache.

Kim et al. [122] consider a system which has both flash memory and SRAM and attempt to minimise the amount of SRAM required by paging small areas of flash into SRAM, rather

than mirroring all of the flash. Their optimiser clusters functions into the same page, and rewrites certain branches to call its page manager, which dynamically copies sections of flash into SRAM as needed. Overall this manages to save an average of 40% of the code memory required with an *increase* in both energy consumption and execution time of 10% and 14% respectively. This study does not see any energy efficiency gains from moving code to SRAM since entire pages of flash are loaded multiple times, and the specific energy models used. The study modelled the energy for a 65nm flash chip, and a 90nm SRAM chip; with these parameters it is possible the flash is more energy efficient than the SRAM. This contradicts our measurements reported in Section 5.4 (Figure 5.11) for when both SRAM and flash are on the same technology node.

A similar approach is taken in [123], attempting to reduce the SRAM usage of synchronous data flow applications. A clustering algorithm based on genetic algorithms is used to decide where to place various portions of the data flow graph, so that related code can be loaded into SRAM at once.

Overlaying functions from flash into RAM is a technique employed by deeply embedded SoCs [124], when regions of RAM have differing performance characteristics. This is typically performed by the developer, and for performance reasons, rather than energy.

Cache locality algorithms. Arranging code and data so that it uses the available cache space will has a large impact on performance and energy. Many studies consider how cache energy can be improved, however require hardware modifications, such as placing the cache in a drowsy state [125], or purely target performance by modifying the code and data layout [126] (which should reduce the overall energy). Other studies use allow the processor to use the knowledge of caches misses to place non-critical instructions into power-efficient, but slower functional units.

In general, deeply embedded SoCs do not have caches, although it is possible that similar techniques could be applied in some cases. For example, the position of code in flash memory has some similar characteristics to that of caches — if code is aligned then it will require less energy. Reusing a code-layout-oriented optimisation may reduce energy in deeply embedded SoCs.

Overall there are a wide range of techniques to reduce energy consumption via the compiler. The majority of these techniques do not perform complex transformations of the program's AST or IR, instead focusing on the context of the executing code — how and where the code is executed. This contrasts with traditional optimisations for performance which minimise the amount of code being executed, or attempt to parallelise it (either via instruction level parallelism or vector instructions).

The existing attempts at finding optimisations can be used to guide the search and creation of new energy optimisations. By contrast finding a new optimisation is inherently difficult, since it either requires extensive measurements, or an analytical approach considering low level bit-flips. The analytical approach is often challenging due to the complexity of the problem, and the high level nature of compiler transformations. The optimisations in the rest of this chapter were found by measuring the target systems under different configurations and examining how the energy behaviour changed.

5.3. Embedded flash memory

Embedded flash memory is typically used in deeply embedded SoCs and is on the same die as the processor. The memory is typically single cycle access, depending on the clock rate of

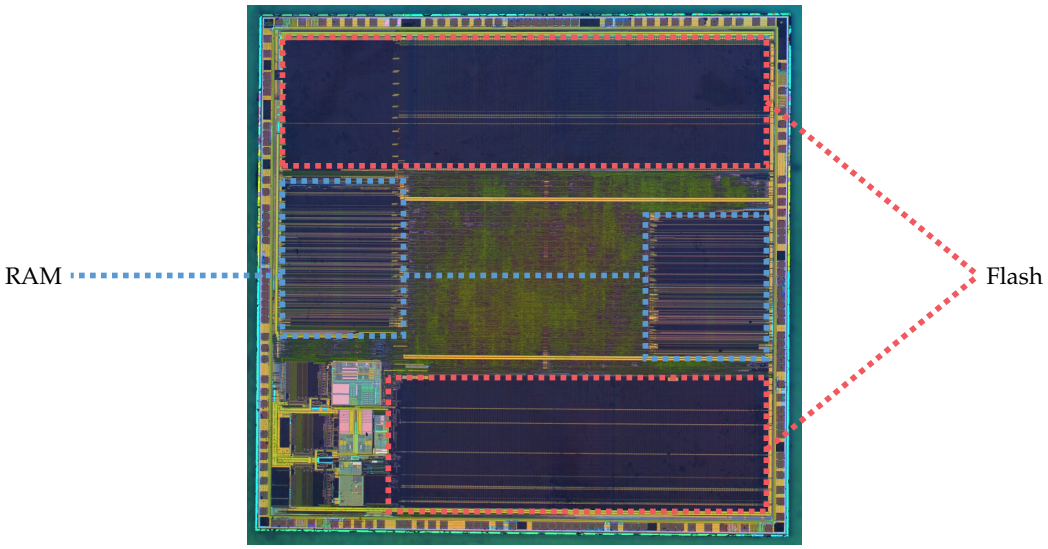


Figure 5.1: Image of the die of a STM32F103VGT6. The SoC is from the same family as the STM32F1, with a larger flash and RAM (1MB and 96kb respectively) [127].

the processor utilising it. Since the flash memory can be large and take up a large percentage of the silicon area, the power dissipation of this component is significant in the overall chip. Figure 5.1 shows an image of the silicon die of a very similar chip to the STM32F1 (differing only in memory sizes).

Embedded flash is typically structured hierarchically, divided into pages, blocks, word-lines and then bit-lines. Figure 5.2 shows a typical arrangement of flash cells. In this figure, the flash memory is divided into pages (shown vertically), then each page is divided into blocks (shown inside the dashed boxes). The address is partially decoded to select which page and block the required memory address is located in. Each block is divided into k word lines, and n bit-lines.

The memory is typically accessed n bits at a time, reading from bit-lines $B_0, ..., B_{n-1}$ of the correct page [128, 129]. To perform the access, each bit-line is precharged to a specific voltage between V_{DD} and GND. The address decoder for the correct block then asserts the control line for the required word-line, and then uses S_b and S_g to connect the set of flash cells to the bit-line and ground, respectively. This has the effect of pulling the voltage on the bit-line up or down, depending on the stored charge in the flash-cell. The sense amplifiers at the end of each bit-line amplify this change and buffer the value onto the data bus, where the data is returned to the processor.

Figure 5.3 shows the effect that alignment in flash memory has on the execution of code for six of the platforms with flash memory. For these tests, both 8-byte and 10-byte loops were aligned to different offsets from the beginning of memory, and their energy consumption per loop iteration measured. For these small loops there is between a 5% and 15% change in energy consumption, depending on the alignment of the code. This is seen on all SoCs except the MSP430FR SoC — this SoC is identical to the MSP430F except it uses FRAM (Ferroelectric RAM) instead of flash as its non-volatile storage. Each of the six platforms shows a different energy consumption profile, although there are common features between SoCs. The common features are lettered **A** – **D**, and explained below.

A Feature **A** highlights the zigzag pattern seen every 4 bytes of alignment — alternating

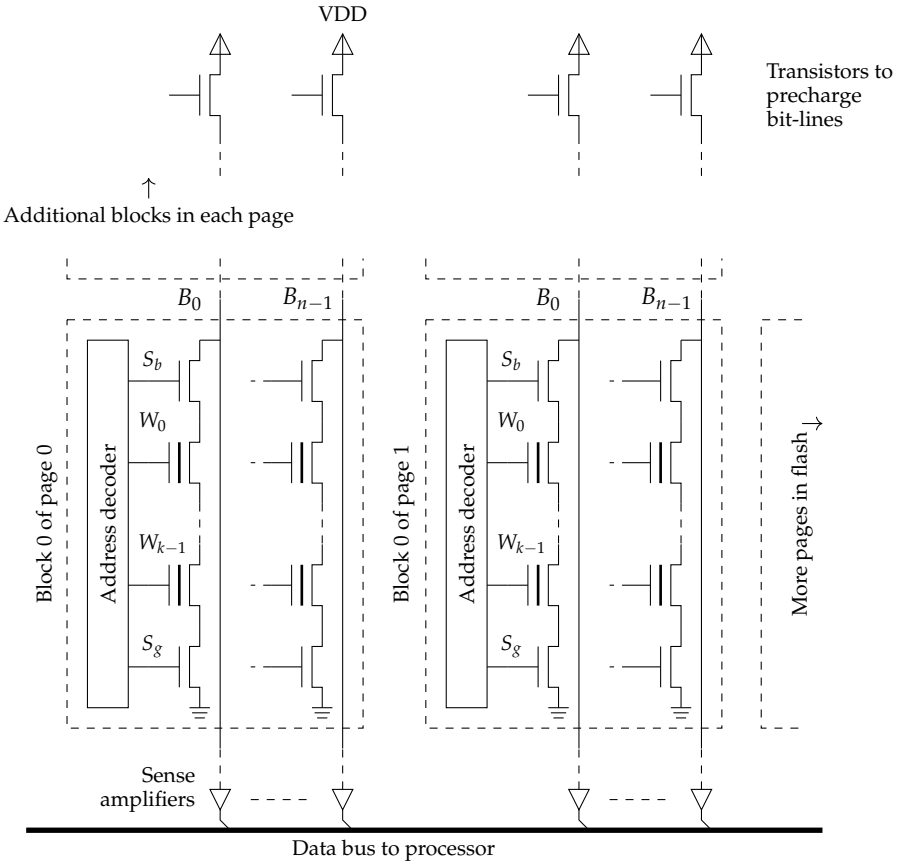
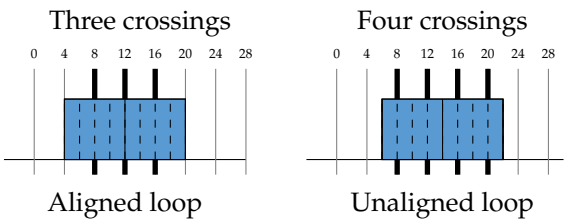


Figure 5.2: The internal structure of embedded flash memory. The bit-lines are labelled $B_0, ..., B_{n-1}$, the word-lines are $W_0, ..., W_{k-1}$, and the switches controlling each block are S_b and S_g.

between high and low energy per iteration. This can be seen for 8-byte loops in the STM32F0 and STM32F1, and the 10-byte loop in the PIC32 and MSP430F. The alignment to a 4-byte boundary greatly affects the energy consumption in three of the platforms and has a smaller effect on one other (PIC32). The effect occurs because the flash has 32 bit-lines, and extra fetches from flash must be performed to support any type of unaligned accesses. In the STM32F0 and STM32F1 SoCs, this is only seen on loops whose size is a multiple of 4 bytes — if the loop is not a multiple then the same number of 4-byte boundaries are crossed whether the loop is aligned or not. An opposite effect is seen in the PIC32 and MSP430F, due to differences in instruction prefetching when a branch occurs. The following diagram shows how unaligned accesses can affect a loop whose size is a multiple of four.



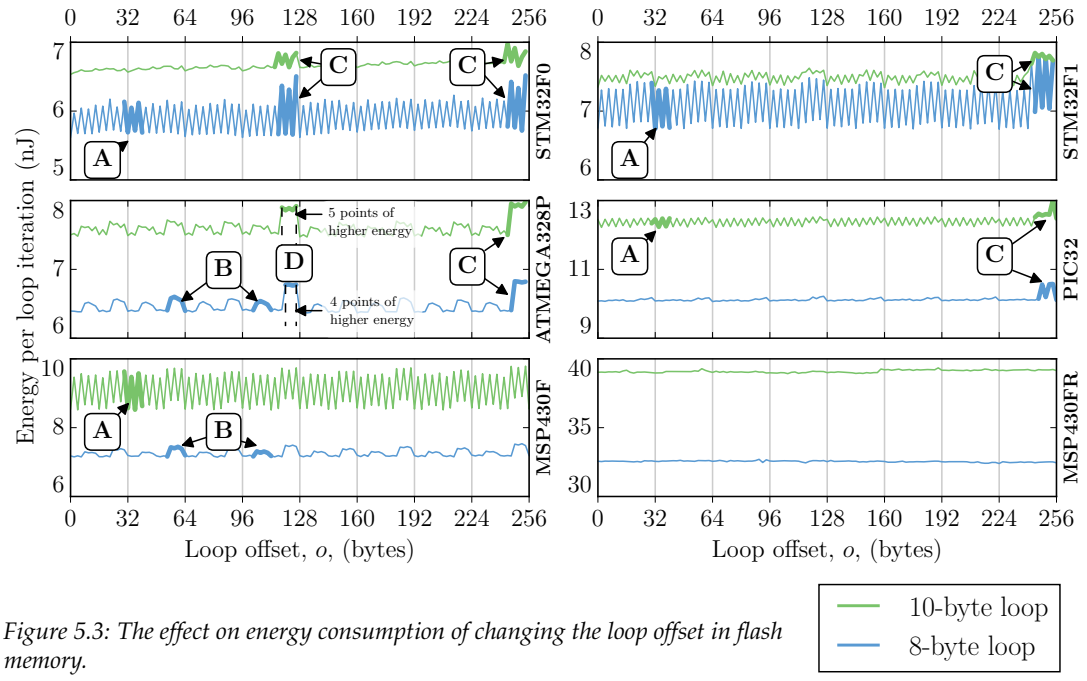
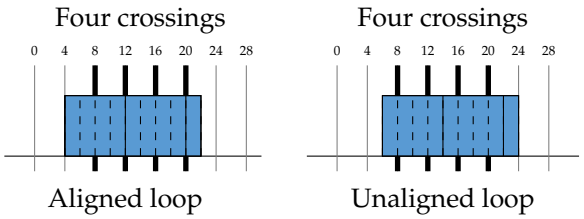


Figure 5.3: The effect on energy consumption of changing the loop offset in flash memory.

When the loop (shown as a rectangle) is aligned fewer 4-byte boundaries are crossed. This is in contrast with the following loop, when the loop's size is not a multiple of four (here, 18 bytes instead of 16).

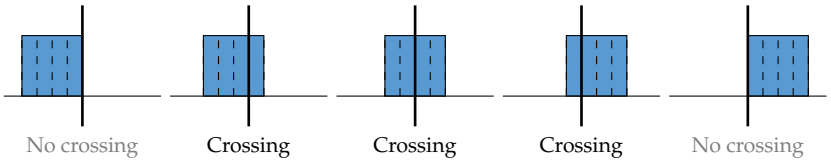


Here, the number of 4-byte boundaries crossed does not depend on the loop alignment. This phenomenon explains the zigzag pattern in energy consumption for these SoCs.

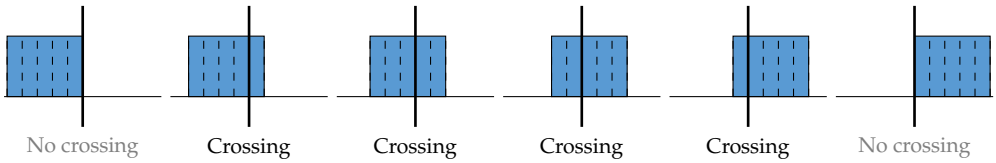
B Some of the processors have effects which modify the energy consumption based on the 16-byte alignment. This is mostly seen in the ATMEGA and MSP430F, where aligning to the boundary reduces energy consumption. A similar mechanism to that of feature **A** is responsible for this — the loop crosses a block boundary at certain alignments, causing additional switching, and energy.

C A further large effect is seen when the loop straddles a page — page boundaries typically occur every 128 or 256 bytes. This is often a significant effect, as changing a page can be a large amount of circuitry to switch.

D Feature **D** illustrates the effect the size of the loop has on the increase in energy. As the size of the loop increases, the number of alignments which can straddle a k-byte boundary increases.



Four instructions, three alignments which straddle the boundary.



Five instructions, Four alignments which straddle the boundary.

The diagram above shows that a loop with n instructions will have $n - 1$ alignments which straddle a k-byte boundary, such as a 128-byte page (assuming $n < k$).

The graph for MSP430FR does not exhibit any of the features found in the other graphs, because the SoC uses FRAM instead of flash. Other than the memory the SoC is identical to MSP430F — the differences in the graphs are a direct result of using FRAM. FRAM (Ferroelectric RAM) does not have the same architecture and can be accessed in a randomly, rather than divided into pages and blocks as with flash. This leads to a flat energy profile.

The variety of features, and large differences between each SoC require a generic way to model the phenomenon, rather than the individual construction of an energy model for each SoC. Each SoC's flash memory energy consumption can be estimated by assigning costs to each 2^k-byte boundary and counting the accesses which transition this boundary. See the following section for details of the model. The model can be used to optimise the energy consumption due to executing instructions from embedded flash (covered in Section 5.3.2).

5.3.1. *Energy model*

This section describes a generic model which can be used to predict the energy consumption when accessing embedded flash memory. The model focuses on memory-read energy, since in deeply embedded systems the flash is rarely written, while reads are frequently performed for instructions and constant data. The details of the structure of the embedded flash are rarely revealed outside the manufacturer, meaning that a generic model must be able to handle many differing flash architectures.

The model is based on the observation that when two memory accesses access different 2^k-byte blocks there will be a circuit state change overhead, as different components are enabled and disabled. For example, an overhead would be incurred when an access is performed in block 0, then a subsequent access is performed in block 1 (see Figure 5.2).

An energy cost, E_k, can be assigned to the change of each 2^k-byte region, starting at memory location 0. For example, if an instruction i_0 is at address $i_0 = 0$ and instruction j_0 is at address $j_0 = 2$, both a 1-byte boundary and a 2-byte boundary will have been crossed. The energy cost for this transition can be represented by:

$$i_0 \rightarrow j_0 = E_0 + E_1, \tag{5.1}$$

where E_0 and E_1 are the costs of crossing a 1-byte and 2-byte boundaries respectively. Similarly, if $i_1 = 2$ and $j_1 = 4$, the energy cost will be:

$$i_1 \rightarrow j_1 = E_0 + E_1 + E_2. \tag{5.2}$$

The hypothesis that each 2^k-byte region can be assigned an energy cost leads to the following equations describing the total energy for a single address transition,

$$i \rightarrow j = \sum_{k=0}^{N(i,j)} E_k, \tag{5.3}$$

$$N(i,j) = \left\lfloor \log_2\left(i \oplus j\right) \right\rfloor. \tag{5.4}$$

In the above equation, $i \rightarrow j$ represents the consecutive memory access from address i to address j. The term, $N(i,j)$, represents the number of regions that have been crossed by that memory access, i.e. the highest bit in the address that has changed. The $\oplus$ operator is the exclusive-or operator.

Equation 5.3 can be composed into an expression containing all the sequential memory accesses, giving the total memory energy consumption for that sequence of accesses, $T = T_0, T_1, \ldots T_{n-1}$.

$$E(T) = \sum_{i=0}^{n-1} \left(T_i \rightarrow T_{i+1}\right). \tag{5.5}$$

In this equation, T_i is the address of the i^{th} of n accesses to the flash memory. The total energy consumption of the access sequence is given by $E(T)$. This forms a generic model which can estimate the energy given an arbitrary sequence of addresses accessed within the flash. When executing instructions directly from flash, the sequence of accesses can typically be divided into 3 components (see Figure 5.4):

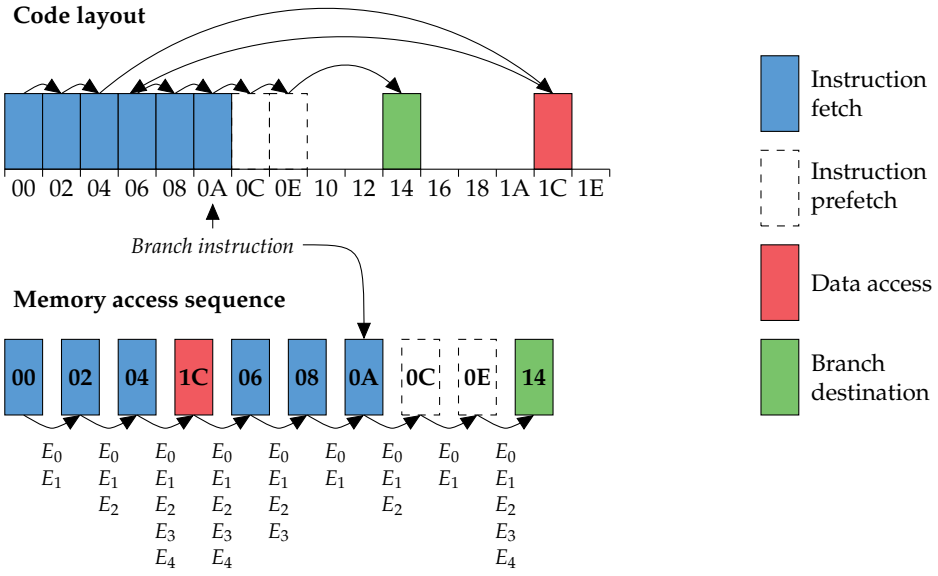


Figure 5.4: An example code memory layout and the ordering of memory accesses it produces.

Fetching the instructions. This can be determined statically within a basic block, since the sequence of instructions is known. The sequence of accesses from instruction fetching, M^I, can be calculated for a basic block, given as,

$$M^I = A(0), ..., A(s-1), \tag{5.6}$$

where s is the number of instructions in the basic block and $A(x)$ gives the address of the x^{th} instruction in the basic block. This equation captures each memory address accessed by sequential transitions inside the basic block.

Implicit prefetching of instructions (typically following a branch instruction). In pipelined processors, the memory fetch for the next instruction happens during the execution of the current instruction (prefetching). However, if the instruction modifies the control flow that memory access may have to be repeated with the correct address. The analysis of which prefetches occur can be done statically for direct, unconditional branches, however, data-dependent conditional branches can branch to two or more locations which may not be known ahead of runtime. The sequence of memory accesses, M^P, from instruction prefetching is,

$$M^P = P(0), ..., P(N^p - 1), B_{dest} \tag{5.7}$$

$$P(k) = S_{insn} \cdot k + A(s), \tag{5.8}$$

where s and $A(s)$ are as given above. $P(k)$ gives the address of the k^{th} prefetched instruction, and N^p is the total number of prefetches. S_{insn} is the size of each instruction fetched, and B_{dest} is the destination of the branch. These accesses can be appended to M^I if the final instruction is a branch, since the $P(k)$ expression is the address of each access, k, after the last instruction in the basic block, $A(s)$.

Data accesses by the instruction stream. If the processor executes a load from flash then this memory access should be accounted for. This is especially problematic to do statically if the processor has a unified address space and a load instruction could reference either RAM or flash. The accesses to data can be included by inserting the data address (if in flash) after the respective instruction fetch in the memory access sequence.

An illustration of the components is shown in Figure 5.4. The top part of the diagram shows the layout of various instructions and the branches between them. The sequence below shows the order in which memory locations are accessed, and the type of operation causing the memory access. The coefficients of the model are below the memory access sequence. Summing these coefficients gives the following cost:

$$9E_0 + 9E_1 + 6E_2 + 4E_3 + 3E_4. \tag{5.9}$$

Overall, Equation 5.5 can be used to estimate the amount of energy consumed by the embedded flash memory. This model can be used for optimisation, or static energy determination in conjunction with an instruction level energy model. The parameters for the model are empirically determined for various chips in the model parameters section (Section 5.3.1).

SoC	Model parameters (pJ)							
	E_2 (**A**)	E_3	E_4 (**B**)	E_5	E_6	E_7 (**C**)	E_8 (**C**)	N^p
STM32F0	300	27	6	0	9	100	6	2
STM32F1	500	0	6	34	4	10	190	2
ATMEGA	0	22	36	27	9	107	24	1
PIC32	225	0	10	18	8	13	113	1
MSP430F	408	0	34	26	15	13	13	1

Table 5.1: Model parameters for the different SoCs. The letters correspond to the features shown in Figure 5.3. Parameters with high energy costs are highlighted. N^p gives the amount of prefetching performed by the processor.

Static whole program estimation

The generic flash model given in Eq. 5.5 relies on a sequence of accesses to the memory which can be difficult to obtain statically. This prevents the use of the model at compile time for optimisation decisions (such as code placement). Components of the access pattern which cannot be determined statically are the locations when loading data from flash and branch destinations. These are addressed below, so that a static model of a program's energy consumption can be created.

The pattern of data locations can be simplified in the model by an observation that many accesses to static data are known at compile-time. Thus, using data-flow analysis, the compiler can often work out whether code is accessing read-only data (in flash) or volatile data (in RAM). If the data access is in flash, then the data can either be stored in the code, in a constant pool, or further away in its own data section. When the data is in a constant pool, it is typically accessed by a static offset relative to the program counter, e.g. with ARM's `adr` instruction. For data which is potentially further away, an estimation of the distance can be fed into the model. As seen from the parameters determined in the next section the largest energy coefficient is often caused by the 256-byte boundary — if the data section can be reasoned to be further than 256 bytes from the current instruction then the energy cost for that transition will be at least $E_0 + ... + E_8$, even if it is much further than 256 bytes away.

An estimation of the energy cost for jumping to the branch destination can also be created. In the majority of cases conditional branches are short-range with two possible destinations. These can either be approximated by just using the common coefficients between the two destinations, or using the compiler's knowledge of the likelihood of the branch being taken to weight the energy. For example, if the last address transition of the branching is $6 \rightarrow 8$ or $6 \rightarrow 0$, the possible energy consumptions are:

$$6 \rightarrow 0 = \underbrace{E_0 + E_1 + E_2} \tag{5.10}$$

$$6 \rightarrow 8 = \underbrace{E_0 + E_1 + E_2} + E_3. \tag{5.11}$$

The two expressions have common coefficients, where the first three terms of Equation 5.10 are also present in Equation 5.11. The average could be used as an approximation if no other information about the branch destination was available. If branching probabilities are available (possibly provided by the compiler, or profiling), the last coefficient (E_3) can be scaled by the likelihood that the destination is 8, rather than 0.

Other branches have unknown destinations, and cannot easily have their energy consumption estimated. Return instructions can be estimated by using the call graph and enumerating likely parent functions. Indirect branches could be completely freeform with any destination. In this case, either a profile or more sophisticated static analysis is necessary to determine possible

branch targets. These cases occur infrequently in embedded programs, and therefore will not significantly increase the error of the energy prediction.

These points can be combined when there is information about the branch destinations into a model for the entire flash memory energy of a program, P. The following equation is split into two parts, summing the energy of the accesses inside each basic block, and summing the energy of the branches between each basic block. Each part is scaled by the number of times either the basic block is executed (F_b) or the number of times a block branches to another ($F'_{i,j}$),

$$P = \sum_{b \in B} E(T_b) \cdot F_b + \sum_{i,j \in B} E(T'_{i,j}) \cdot F'_{i,j}, \tag{5.12}$$

where $E(X)$ gives the energy consumption of a stream of flash accesses, X, as in Equation 5.5. In the first part, T_b gives the expected sequence of accesses, except for the branch destinations for the basic block b, out of the set of all basic blocks in the program, B. The energy of these accesses is scaled by the number of times the basic block is executed, F_b. In the second part, $T'_{i,j}$ is the sequence of accesses caused by the branch at the end of block b_i to the start of the block b_j. The energy is scaled by the number of times each edge is taken, $F'_{i,j}$.

The stream of accesses is defined,

$$T_b = \overbrace{A_b(0),\ \ldots,\ A_b(s-1)}^{M^I},\ \overbrace{P_b(0),\ \ldots,\ P_b(N^p-1)}^{M^P}, \tag{5.13}$$

where $A_b(x)$ gives the address of the x^{th} instruction in block b (with s instructions in it), and $P_b(x)$ gives the address of the x^{th} prefetched location as per the model. These are formed from the accesses from instruction fetch and instruction prefetching, M^I and M^P respectively. The stream of accesses for the branching behaviour of the basic block is given,

$$T'_{i,j} = P_i(N^p - 1),\ A_j(0), \tag{5.14}$$

where this is intuitively the last access performed by basic block b_i and the first performed by b_j. For the example given in Figure 5.4, the data access is ignored (due to being difficult to determine statically), and the accesses split between T_b and $T'_{i,j}$,

$$T_b = \texttt{00}, \texttt{02}, \texttt{04}, \texttt{06}, \texttt{08}, \texttt{0A}, \texttt{0C}, \texttt{0E} \tag{5.15}$$

$$T'_{i,j} = \texttt{0E}, \texttt{14}. \tag{5.16}$$

By formulating a static model of a program's flash energy consumption, optimisations can use this information to make positioning and alignment decisions requiring only information that is already present in the compiler or at the assembly level, such as the control flow graph and estimates (or profile) of execution counts for basic blocks and edges.

Estimating the model parameters

Linear regression was used to find the value of the parameters to the model for five SoCs. For each SoC, a large number of tests were measured for loops with different sizes and alignments. The set of alignments explored, O_{loop}, and the set of loop sizes used, S_{loop} are,

$$O_{loop} = \{0, 2, 4, \ldots, 256\} \tag{5.17}$$

$$S_{loop} = \{8, 10, 12, 14, 16\}. \tag{5.18}$$

```
1       b  loop
2       .align 8            ; Align to 256 bytes
3       .fill  o, 1, 0      ; o ∈ O_loop
4   loop:
5       .repr s/Sinsn - 2   ; s ∈ S_loop, Sinsn = S_insn
6           nop
7       .endr
8       sub r0, #1
9       bne loop
```

Figure 5.5: Example test to exercise different sized loops, $s \in S_{loop}$, at different alignments, $o \in O_{loop}$.

SoCs	NRMSD (%)
STM32F0	27.1
STM32F1	17.5
ATMEGA	5.6
PIC32	8.4
MSP430F	17.7
Mean	13.2

Table 5.2: Cross validation results for all SoCs.

Program	NRMSD (%) STM32F0	NRMSD (%) ATMEGA	Features B_u	Features B_c	Features BB
I	14.1	8.8	1	2	4
II†	19.3	6.1	1	2	3
III†	18.8	9.1	1	2	3
IV	14.3	5.7	2	3	5
V‡	21.7	7.4	1	1	2
VI‡	9.5	7.6	1	1	2
Mean	15.7	7.3	-	-	-

Table 5.3: Validation results using complex loops, with features: B_u, number of unconditional branches, B_c, number of conditional branches, and BB number of basic blocks.

†‡ Pairs of tests with the same structure, but different sized and aligned blocks.

All the instructions used in these tests were 2 bytes and the loop bodies were `nop`s to minimise the processor's impact on the test results (see Figure 5.5). Using this set of tests allows the exact access sequence to the flash memory to be found analytically (infrequent conditional branches, no data access to flash), and enough tests to allow linear regression to correctly estimate the parameter values.

The values of the parameters for each SoC are shown in Table 5.1, with large parameters highlighted. In particular, the 4-byte boundary (parameter E_2) is often associated with a large energy cost, suggesting that these SoCs have embedded flash with 32 bit lines. This also corresponds to the feature **A** in Figure 5.3.

The other highlighted parameters are the 128- and 256-byte boundaries (parameters E_7 and E_8). These are likely the page sizes of the respective SoCs. Parameters E_4 and E_5 have large values in two of the SoCs (ATMEGA and MSP430F) which correspond to feature **B** in Figure 5.3 and are caused by the block structure of the flash.

Model validation

Two forms of validation were performed on the model, cross validation and then testing on unseen, more complex loops. Cross validation was used to validate the parameters of the model, and repeated for all combinations of datasets for each SoC. The Normalised Root Mean Square Deviation (NRMSD) metric is used to give an error percentage for the entire set of alignments. The metric compares all corresponding points for a given loop size, and measures the total error between two series. This gives a better indication of the error than an individual error, since

optimisation decisions can still be made if the prediction is not absolutely correct, but correct relative to the points before and after it.

The cross validation tested the model was sufficient by finding the model parameters with $|S_{loop}|-1$ of the loop sizes, and using the final set of alignments to test the model. For example, one cross validation test would find the parameters using $S_{loop} = \{8, 10, 12, 14\}$, and evaluate the model on $S_{loop} = \{16\}$, ensuring that the data used to find the parameters was not used to test the model. Table 5.2 gives the average NRMSD for each platform using each combination of cross validation tests.

The error is very low for ATMEGA and PIC32, since the simple pipelines allow the exact memory access sequences to be modelled easily. Thus, there are very few memory accesses which are not accounted for by the model and the realised error is attributed to measurement noise and effects the model does not capture. Two other SoCS, MSP430F and STM32F1 have a slightly higher error, although the error is still low enough to make optimisation decisions. The additional error is due to memory accesses which were not taken into account. For example the STM32F1 uses an extra prefetch buffer which will affect the locations of flash memory accessed. Note, this is different from the prefetching of instructions accounted for by the model. The prefetch buffer in the STM32F1 attempts to fetch extra instructions, and to prevent stalls in the pipeline, however, full implementation details are not available and cannot be added into the model. The STM32F0 has the largest error, again due to a prefetching scheme which causes unknown memory accesses — larger compared to the STM32F1 which has branch prediction informing the prefetch buffer.

The ATMEGA and STM32F0 SoCs have the best and worst errors, respectively and further extensive validation was performed on these SoCs. This was performed using more complex loops with multiple basic blocks and branching. These loops, numbered **I** – **VI**, have conditional branching as well as multiple basic blocks inside the loop. The number of conditional and unconditional branches, and basic blocks are shown in Table 5.3. The features show a range of control structures within the loop, representing the behaviour typical programs would have. The exact control flow graphs of the complex loops can be seen in Figure 5.6. This diagram shows the alignment and branching of each basic block in the loops. The error for the ATMEGA is slightly higher than in the cross validation tests. This is due to the extra conditional branches adding energy coefficients which are not accounted for (see previous Equations 5.10 and 5.11). The conditional branches cause the opposite effect in the STM32F0 — a lower error is achieved. The high error seen in the cross validation tests is caused by the prefetch buffer fetching additional memory locations. With more complex loops, these prefetched locations correspond to subsequent instruction accesses which *are* captured in the model, hence lower error is achieved.

Figure 5.7 shows a comparison between the predictions of the model and measured results of the loop **V**. The STM32F0 graph shows that the model follows the trend of the measured results, despite the comparatively high error. While the magnitude of the peaks and troughs is not correctly predicted, the direction of change is correct, along with the significant increases (such as at the end of a page). The prediction for ATMEGA is much closer to the actual energy consumption, with the graph having a similar shape. The source of the error for this SoC is due to the under estimation of energy near the 128-byte boundary.

5.3.2. *Optimisation*

An optimisation can be formed with the embedded flash energy model, using it to inform the compiler about good positions in memory for code. The total flash energy consumption should be lowered if frequently executed basic blocks are aligned in such a way that they minimise the

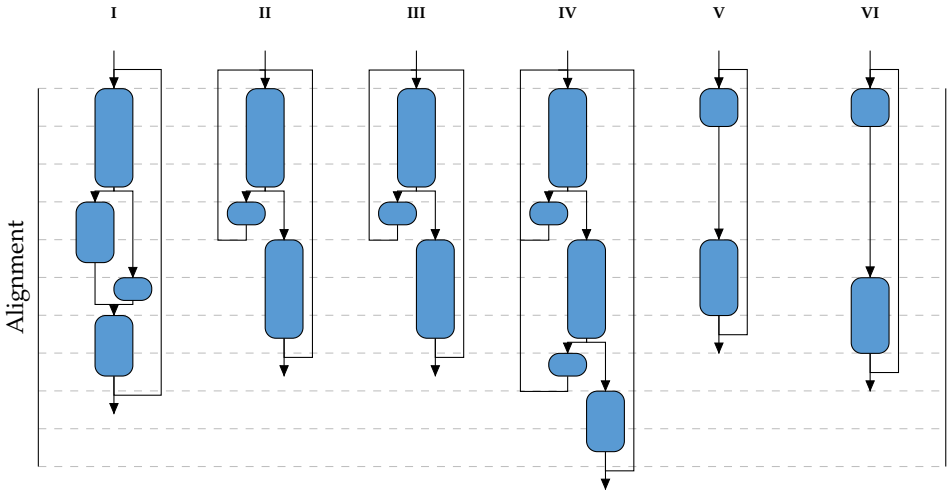


Figure 5.6: The basic blocks structure and their alignments for the complex loop tests.

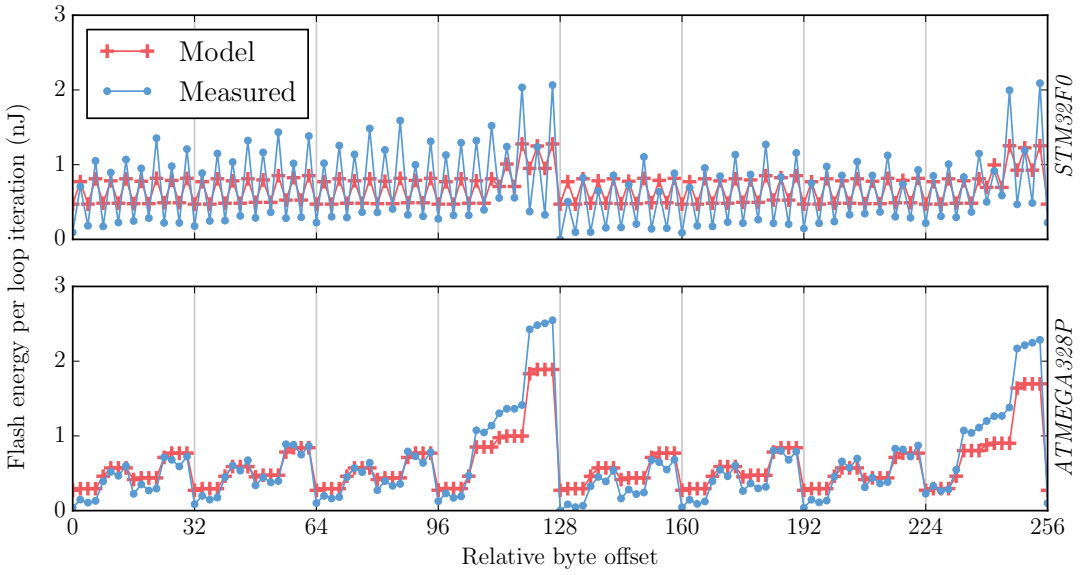


Figure 5.7: Comparison between the model predictions and measured results for STM32F0 and ATMEGA.

number of expensive region crossings, as informed by the model. Three possible optimisations of varying complexity are outlined below.

Loop alignment. Ensuring that important loops are properly aligned is a possible optimisation. The optimisation would reduce energy consumption by understanding the structure of embedded flash, without necessarily using the model explicitly. For many of the platforms the greatest model parameter is for the 4-byte region (E_2). The energy consumption caused by this parameter can be reduced by ensuring loops are aligned to a 4-byte boundary. This optimisation is often seen in modern compilers for performance reasons — 4 bytes is the minimum alignment possible for many processors' memories [70], and unaligned accesses often have a performance penalty or are not supported. Alignments at higher boundaries have not been previously considered, as there is often less or no performance (execution time) benefit.

An energy saving transformation based on aligning loops to a particular alignment boundary should consider the following items:

- **Estimated minimum number of iterations of the loop.** A trade-off must be made between the cost and the benefit of aligning the loop. This trade-off will be affected by the number of iterations for which the loop is executed.
- **Size of the loop.** The transformation should consider the size of the loop, because large loops will have a lower relative decrease in energy consumption, compared to smaller loops.
- **Space wasted to align the loop.** When aligning the loop to a k byte region, up to $k - 1$ bytes may be wasted. The wasted space must be balanced against the benefit of aligning the loop, since blindly aligning every loop to a large boundary could cause a significant increase in code size. It is possible to minimise this by moving infrequently executed basic blocks into the space before the loop.
- **Loop entry distance.** The performance and energy costs of branching into the loop must be weighed against the cost of padding the offset with `nop`s.

Basic block alignment. An alternative optimisation is to align each basic block to reduce the number of transitions. The optimisation inserts an amount of space before certain basic blocks, changing the alignment of that basic block (and subsequent basic blocks). This optimisation is more complex than the previous optimisation, since there is the cascade effect of the alignment affecting the following basic blocks. However, it has the potential to be more effective than purely aligning loops, since it considers basic blocks individually and still can align blocks inside of loops.

Basic block reordering. Just aligning basic blocks may not be able to fully reduce the energy consumption, and possibly could increase the space required by a large amount. Allowing full flexibility in the placement of a basic block, with arbitrary positions in memory, is the most likely optimisation to reduce energy consumption, however, it is the most complex. Constraints must be applied to the placement to ensure that two basic blocks do not overlap, and that the branches at the end of each basic block have enough range to jump to their targets. Additionally, some of the branches may have to be rewritten — if the fall through target of a branch is moved, an additional branch must be insert which both increases the energy consumption and execution time. This optimisation is not implemented due to its complexity.

Benchmark	O0 (%)	O1 (%)	O2 (%)	O3 (%)	Os (%)
2dfir	22.8	23.9	23.9	24.1	23.9
blowfish	16.0	16.5	18.5	18.6	17.5
crc32	18.2	18.9	19.0	18.1	18.3
cubic	21.9	21.7	21.8	21.8	21.8
dijkstra	14.1	17.7	17.4	17.8	16.8
fdct	14.1	19.4	18.5	18.3	18.7
matmult-float	22.7	23.6	23.8	23.7	23.2
matmult-int	18.6	19.8	17.3	17.0	20.1
rijndael	15.5	19.5	18.9	19.4	19.2
sha	19.0	19.0	18.7	18.9	18.7
Average	18.3	20.0	19.8	19.8	19.8

Table 5.4: The proportion of the program's energy consumption which is taken by crossing flash boundaries.

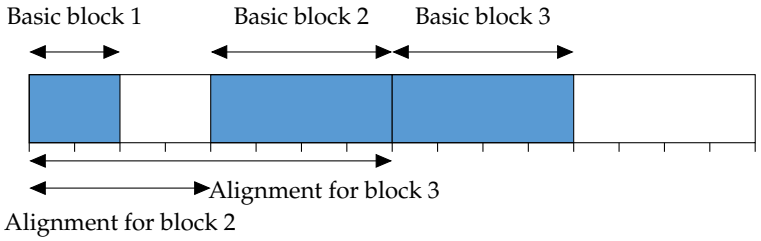


Figure 5.8: Alignment example of three basic blocks

The following sections explore these optimisation concepts, evaluating their potential impact on the energy of the benchmarks in BEEBS. As an initial reference point, the static program model (Equation 5.12) was applied to each benchmark's trace at each of the main optimisation levels. This gives a strict upper bound for the amount of energy that could be saved by these optimisations. Achieving this bound is not possible, since the resultant program would be required to perform zero changes in memory access, and thus all be stored in the same location.

Table 5.4 shows the absolute and relative best possible savings. These figures were gathered from an instruction trace of each benchmark on the STM32F1 SoC, and applying the energy model. A significant proportion of the overall energy consumption of the SoC is consumed by address changes in the flash. While the energy consumption is large, it is likely that only a fraction of it can be reduced due to unavoidable accesses.

Implementation and evaluation

The potential optimisations were implemented in various ways. The first optimisation was implemented[1] in GCC [41], however did not succeed in reducing energy on the ATMEGA. The optimisation identified loops, and inserted padding before the loop so that it was aligned to a larger boundary. The additional padding is formed of `nop`s, and must be executed, negating any energy savings that the loop alignment achieved.

The second possible optimisation using the model attempts to insert an amount of space before each basic block aligning to a different boundary. Due to the non-linear nature of the model, quickly finding an optimal set of alignments for all the basic blocks in the code is a

[1] Thanks to Jörn Rennecke of Embecosm for the implementation in GCC.

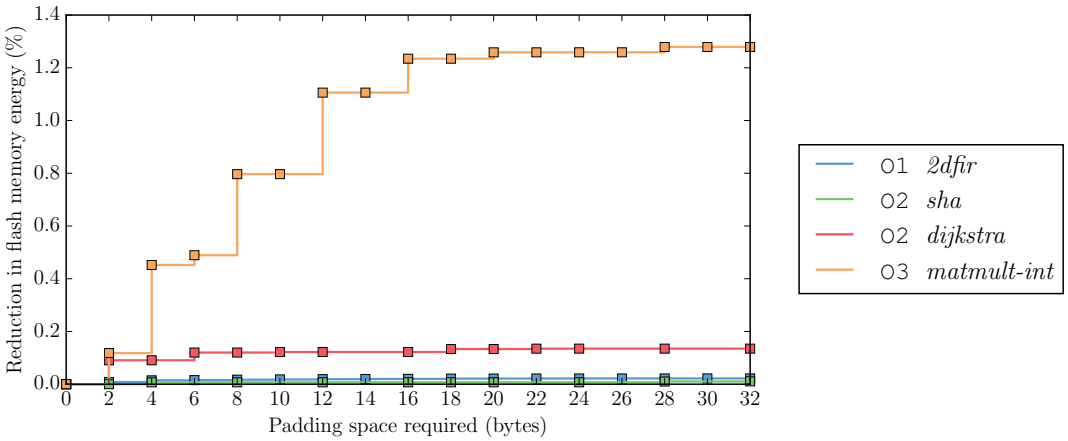


Figure 5.9: Pareto frontiers of the energy and space trade-off for aligning basic blocks (STM32F1).

difficult task. Heuristics and iterative evaluation of the model can be used to guide the positions of the basic blocks and devise an acceptable solution. Figure 5.8 shows three basic blocks, with basic block 2 aligned to a 4-byte boundary so that it does not straddle this region and cause additional energy consumption. By aligning the second basic block, this changes the alignment of the third basic block such that it does not need additional alignment.

The optimisation was implemented as a post-compiler pass on the ATMEGA and STM32F1 SoCs. The basic blocks, along with their size and alignment were extracted. The static program energy model (Equation 5.12) was encoded into SMT (Satisfiability Modulo Theories) constraints with an offset parameter at the beginning of each basic block. The SMT solver could then find a set of alignments, o_b that minimised the energy consumption of the program, following Algorithm 5.10. This algorithm iteratively decreases an energy value, e, and gives this as a constraint to the solver, which returns a set of alignments that satisfies the constraints. The energy value is decreased until the solver cannot find a solution. The process is repeated with increasing amounts of total space available for alignment (o_{max}), giving the Pareto frontier of energy consumption and additional space required.

Examples of several of the Pareto frontiers generated are shown in Figure 5.9, for the STM32F1 SoC. The graph shows the percentage of possible savings (see Table 5.4) that can be saved by realigning the basic blocks, with up to 32 bytes of extra padding. These savings are not large, and likely to be lost in measurement noise if applied to the benchmarks.

The optimisation does not manage to save significant energy, due to the flash memory model parameters — the largest region-crossing cost is the 4-byte boundary, which is difficult to reduce by aligning the basic blocks. This can be demonstrated by looking at the number of times each boundary is crossed for the unaligned (before) and aligned (after) benchmarks. The following table shows number of transitions across each 2^k-byte boundary for the *matmult-int* benchmark.

		E_2 = 500	E_3 = 0	E_4 = 6	E_5 = 34	E_6 = 4	E_7 = 10	E_8 = 190
Transitions	O3	3,190,939	1,877,418	1,207,977	1,128,960	1,075,239	51,294	51,251
	Realigned	3,201,378	1,882,564	1,234,736	180,542	53,862	51,291	51,221
	ΔTransitions	10,439	5,146	26,759	-948,418	-1,021,377	-3	-30
	ΔEnergy (μJ)	**5.2**	**0.0**	**0.2**	**-32.2**	**-4.1**	**0.0**	**0.0**
		Increase				Decrease		

for $o_{max} = 0, 1, \ldots$ **do**
  e = apply model to find base cost (no change in alignment);
  repeat
    Decrease e;
    Solve for o_b with the following constraints:

$$\sum_{b \in B} o_b \leqslant o_{\max} \quad \text{The sum of the alignments of each basic block is below } o_{max}.$$

$$P < e \quad \text{The program's energy consumption is below } e.$$

  until *the solver returns* `unsat`;
  Record the last found alignment values, along with their energy;
end

Figure 5.10: Algorithm to find the optimal basic block alignment by iteratively reduce the energy consumption.

The optimisation does not manage to reduce the number of transitions for the E_2 parameters, which has the highest cost (500 pJ per transition), instead increasing the amount of energy consumed slightly. However, the solver negates this effect by reducing the number of transitions across the 32-byte boundary, which has both a large cost and large number of transitions, resulting in a small net decrease in energy.

When applied to the ATMEGA, different results are expected, since the SoC does not have a large coefficient for the 4-byte boundary and hence the overall transition energy should be able to be reduced more effectively. However, the saving achieved is also small for this platform. The solver is more effective at reducing the flash memory energy, however, the fraction of energy consumed by this particular SoC is much lower and therefore the savings are actually smaller. The table below shows the changes in region-transitions for the *dijkstra* benchmark, on the ATMEGA.

		$E_2 = 0$	$E_3 = 22$	$E_4 = 36$	$E_5 = 27$	$E_6 = 9$	$E_7 = 107$	$E_8 = 24$
Transitions	`Os`	20,094,489	9,737,031	5,231,518	2,727,340	1,289,695	782,321	352,566
	Realigned	13,769,163	7,702,510	3,630,434	1,760,276	912,483	583,525	210,656
	ΔTransitions	-6,325,326	-2,034,521	-1,601,084	-967,064	-377,212	-198,796	-141,910
	ΔEnergy (μJ)	**0**	**-4.5**	**-5.8**	**-2.6**	**-0.3**	**-2.1**	**-0.3**

Decrease

As with the STM32F1, the majority of the transitions are across the lower boundaries. The ATMEGA manages to significantly reduce the number of 4-byte transitions, even though this does not decrease the energy. This is achieved because the AVR (ATMEGA) only prefetches one instruction ahead ($N^p = 1$), compared to the Cortex-M3 (STM32F1) prefetching two instructions ($N^p = 2$). The transitions across 8-, 16- and 32-byte boundaries also achieve significant reductions, and do manage to reduce the energy consumption but only by a very small amount.

Conclusion

None of the optimisations managed to save a significant amount of energy for any of the benchmarks, despite managing to align the basic blocks and minimise the amount of energy as predicted by the model. In light of this, the maximum potential saving achievable for each benchmark were computed (on STM32F1), shown in Table 5.5. These numbers were found by eliminating all model parameters except E_2, the crossing of a 4-byte boundary, and are

Benchmark	O0 (%)	O1 (%)	O2 (%)	O3 (%)	Os (%)
2dfir	1.4	1.5	1.5	1.5	1.4
blowfish	0.7	0.7	1.1	1.1	1.1
crc32	0.8	0.9	1.0	0.9	0.9
cubic	0.9	0.7	0.7	0.7	0.7
dijkstra	0.8	1.2	0.8	0.8	1.1
fdct	0.3	0.5	0.4	0.4	0.4
matmult-float	1.1	1.1	0.9	1.0	0.9
matmult-int	0.9	0.5	0.1	0.6	0.4
rijndael	0.4	0.8	0.6	0.6	0.7
sha	0.7	0.7	0.8	0.8	0.7
Average	0.8	0.9	0.8	0.8	0.8

Table 5.5: Available savings, assuming that the 4-byte boundary cost cannot be significantly minimised.

the percentage of energy attributable to other regions being crossed. The crossing of 4-byte boundaries is the most frequent, due to consecutive instructions crossing it frequently and thus is the most difficult to minimise.

This difficulty in minimising energy likely applies to the majority of the platforms tested, since the E_2 parameter is often large, and suggests that it will be impractical to optimise a SoC's flash memory energy if there are large model coefficients for small region crossings. Only for the ATMEGA is the parameter small (0), but for this platform the overall energy that could be saved by aligning code is low.

A further complication with the alignment methods is that additional code must be inserted — this takes extra time and thus energy. With platforms that have low coefficients for the smaller region crossings, such as the ATMEGA, the code can be aligned easier, however the average amount of space that needs to be inserted is larger (larger boundaries to align to).

Currently these obstacles make the exploitation of this model difficult, particularly as the amount of potential energy savings are low in many platforms and difficult to achieve reliably.

5.4. RAM Overlay

As mentioned in the previous section (Section 5.3) deeply embedded SoCs typically execute their code directly from flash, while reserving the RAM for runtime data. However, in systems with a unified address space, it is often possible to execute code from the RAM as well. This can be beneficial, since flash typically consumes a large amount of energy when accessed — RAM is usually much less power hungry.

This section discusses how code can be moved to RAM before execution, resulting in lower total energy consumption. Figure 5.11 shows a comparison of the power dissipation for different instructions, executing out of both RAM and flash. In all cases the execution of instructions from flash takes more power than RAM, and in all cases except *load from flash*, the reduction is significant (between 35% and 60% lower). In the last case, the power is reduced, but by very little because the flash still needs to be accessed. Since both flash and RAM are single-cycle access (with the execution time remaining constant), this could form the basis of an optimisation that targets energy consumption via reducing the average power.

The lower power when executing from RAM suggests that significant energy would be saved if the program code was moved into RAM. The savings are exemplified in code which executes a memory copy out of flash or out of RAM. The energy consumption and execution time of the

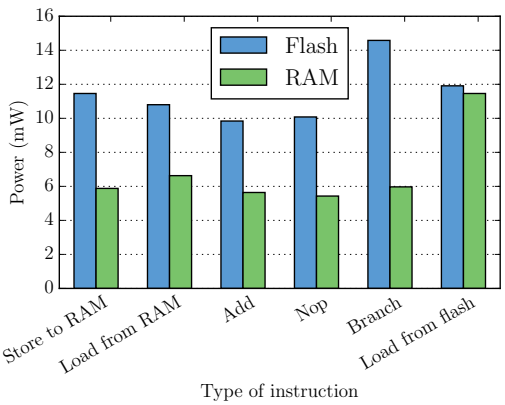


Figure 5.11: Power dissipation of STM32F1 with different types of instructions executing out of flash or RAM.

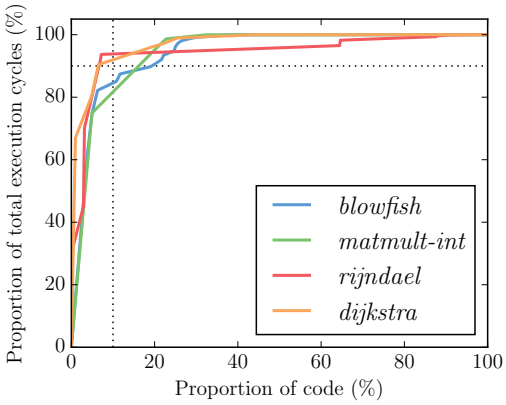


Figure 5.12: The figures shows that only a small amount of code is responsible for the majority of execution cycles in a benchmark.

memory copy are measured:

Code location	Total energy	Total time
Flash	7.7 mJ	750 ms
RAM	5.8 mJ	880 ms

The energy consumption of the benchmark is reduced by 25%, whilst execution time is increased by 17%. This increase in execution time is a result of contention on the RAM — both instructions and data must now be fetched from RAM, and also affects the energy consumption. In a benchmark which did not access RAM so frequently the energy consumption savings may be larger than 25%.

The benchmark placed the entirety of the memory copy function in RAM. Deeply embedded SoCs have much less RAM than flash, typically about eight times less RAM than flash, making copying all of the sections of code impossible. Choosing a subset of the code to place into RAM allows most of the potential energy savings to be achieved, while still remaining within the limits of the spare RAM.

Only a very small number of basic blocks are executed frequently. The graph in Figure 5.12 demonstrates this phenomenon, plotting the proportion of code against the proportion of the total execution cycles required by that code. In most programs just 10% of the code accounts for 80–90% of the total execution cycles (as seen in Figure 5.12). This argument follows for individual functions — placing an entire function into RAM would likely waste the majority of space due to most of the code not being executed frequently. The developed optimisation analyses each basic block in the application, and determine whether or not it should be moved into RAM to save energy consumption, by exploiting the lower power nature of the RAM. It is necessary to consider the code on a basic-block level, rather than on a per-function basic due to the limited amounts of RAM.

5.4.1. *Implementation*

There are several challenges associated with moving individual basic blocks into RAM, instead of residing in flash. These must be solved efficiently to make the optimisation viable. An

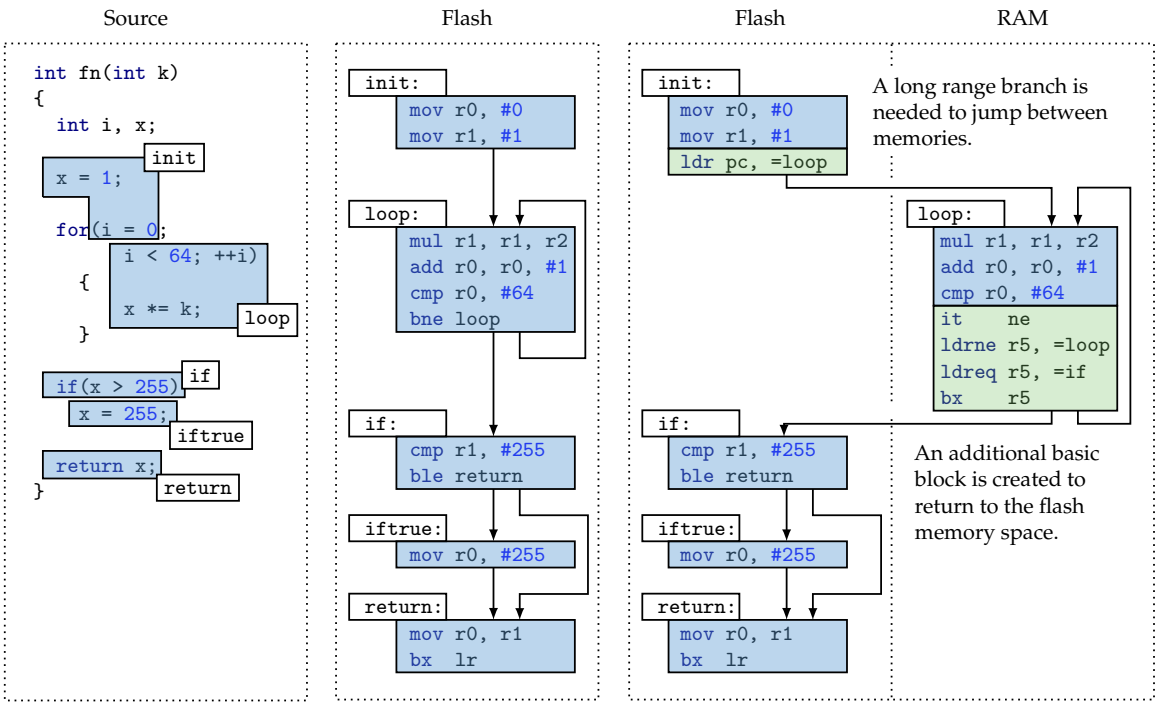
example application of the optimisation is given in Figure 5.13.

Distance between address spaces. In many embedded systems, the flash and RAM address spaces are distinct and numerically far away from each other. For example, in the STM32 SoCs, the flash is typically located at `0x08000000`, whereas the RAM is located at `0x20000000`.

It becomes a problem to jump between the two when relative branches are used (the most prevalent type of branch in embedded code) since there is typically a limited branch range. To jump from `0x08000000` to `0x20000000`, the size of the relative destination field in the branch instruction must be at least 29 bits. Very few embedded instruction sets can do this, therefore all the relative branches must be transformed into absolute branches or indirect branches.

Instrumenting the basic blocks. The branches in the basic block must be rewritten. Additional code may also need to be added, because if a branch is conditional, the execution can 'fall through' into the following code. A branch to the next basic block must be inserted, if the following basic block is not in the same memory.

Rewriting relative references. Some instructions in the basic blocks contain relative references to data or code. In the case of data, the references must be rewritten to point to the data in flash, or have the data copied into RAM as well. References to code are typically relative function calls, which must be rewritten to use absolute function calls. References to data



(a) Example code.

(b) Control flow graph produced from the code.

(c) Optimised control flow graph with code in RAM.

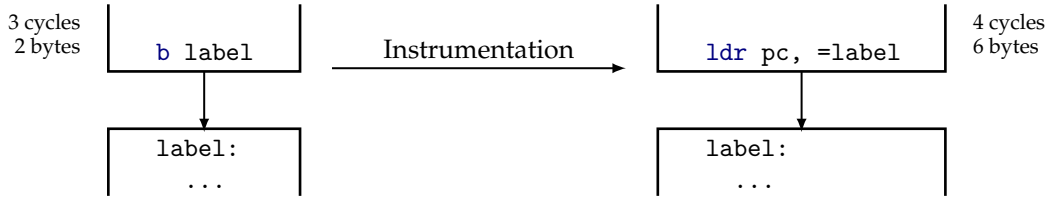
Figure 5.13: An example of how the loop inside a function could be moved into RAM in the STM32F1 SoC.

usually use constant pools. The constant pools can be copied along with the code, and the references updated.

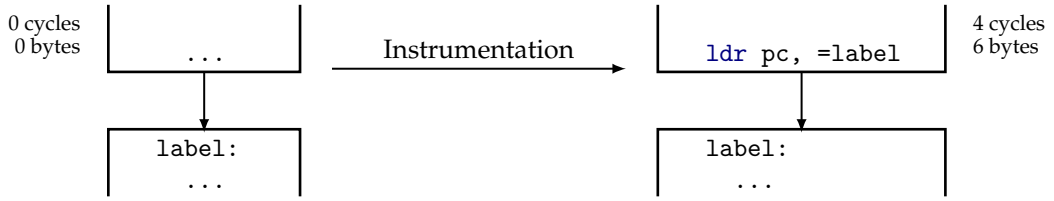
Figure 5.13 gives an example of how only the necessary basic blocks would be placed into RAM, and called directly from flash. The jumps between memories incur additional time, space and energy costs, which must be outweighed by the benefit of placing the basic block into RAM. The trade-off can be made by modelling these constraints, then finding the set of basic blocks which should be in RAM to minimise the overall energy consumption.

For the STM32F1, the optimisation is implemented as a post-compiler pass, modifying the section that a basic block appears in. If the basic block should be in the RAM, it is placed into a section which is loaded into the RAM at startup, at the same time as volatile data. The branches at the end of the instrumented region are rewritten to load the address of the target basic block into a register and jump to it. The rest of this section focuses on STM32F1, however, the method is transferable to other SoCs with minimal modifications.

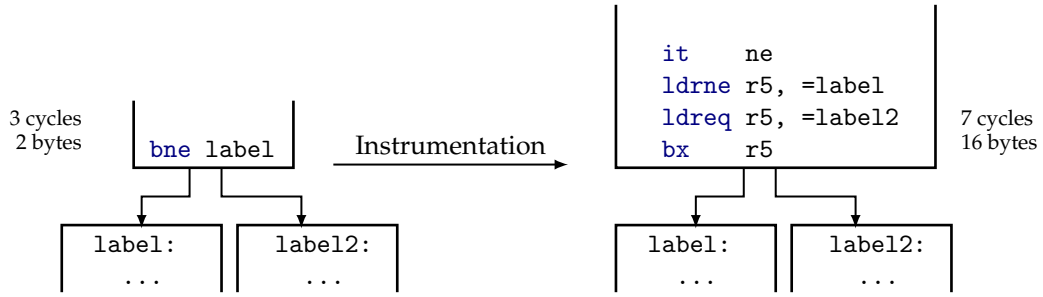
The exact transformations depend on the branch at the end of the basic block. The most simple case is an unconditional branch to the next basic block. The branch gets transformed into an indirect load of the next block's location, as seen below.



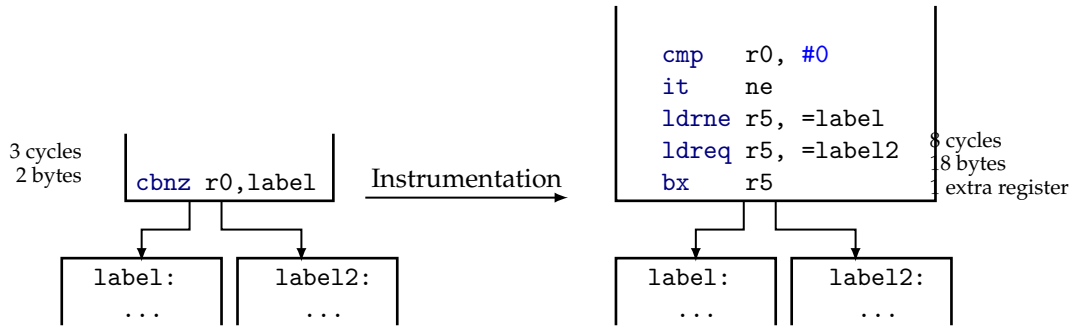
A similar transformation is applied when the basic block ends, but there is no branch (i.e. a fallthrough into the next piece of code). The transformation simply adds a new branch at the end of the block.



The transformations for conditional branches are more complex, since they require loading different addresses based on the condition. This is achieved using the `it` instruction to conditionally load the address of the next instruction in a register and branch to that register.



The final type of branch combines the compare and conditional branch into a single instruction. To instrument this instruction it must be split into separate instructions so that the addresses of the basic blocks can be loaded.



Overall these transformations allow a basic block with any kind of branch at the end to jump to a basic block in a different memory space, for varying costs in terms of cycle count and space. These transformations require the use of a temporary register to enable the branching behaviour. This was achieved in the implementation by preventing GCC from allocating a specific register with the `-ffixed-r5` flag. This has a minor impact on the performance of the code (a maximum of 4.7% increase in energy) however, in a fully integrated version of the optimisation the register could be chosen based on which registers are unused to avoid the penalty. Despite this 4.7% additional cost, energy savings of up to 26% are still achieved.

5.4.2. *Program energy model*

The energy consumption of the entire program can be modelled at a high-level, considering the energy of the memory that code is executing from, along with the overheads of placing that code in the particular memory. For an accurate estimation of the energy consumption, input data values to the program would need to be known. However, an estimation of the number of times a section of code is expected to execute provides similar results to having full information (see Section 5.4.3).

Model parameters

There are several parameters to describe each basic block, the connectivity between basic blocks and constraints specified by the developer (such as the maximum amount of code that can be moved into RAM). All of the parameters can be derived or estimated statically from the code or

the CFG, and are listed on the current page.

Derived basic block parameters

- S_b The size (bytes) of the basic block, b, as compiled into flash.
- C_b An estimate of the number of cycles taken to execute the basic block. Often this is an estimate due to pipeline complexities, and fetch times being dependent on code alignment. Additionally, this includes the number of cycles for any branch at the end of the basic block, which may be different based on the direction taken.
- F_b The number of times the basic block is executed. An accurate number can be assigned to this parameter if the application is profiled. Otherwise, a simple estimate can be used to guide the optimisation.
- $Succ(b)$ A set of basic blocks which are the immediate successors to the block, b.

Instrumentation overheads

- K_b The space overhead when instrumenting a basic block to jump between memories (in bytes). This is both the extra instructions added, plus any data that those instructions require (such as the address of the next basic block).
- T_b The time overhead when instrumenting a basic block to jump between memories (in cycles).

RAM overheads

- L_b The time overhead when executing from RAM. Additional cycles may be required if a basic block is in RAM, due to contention on the memory bus.

There are other parameters which are general inputs to the model, rather than derived from the code. These parameters are either specific to the target SoC (found by characterising the specific chip) or specified by the developer.

Hardware parameters

- E_{flash} The energy cost of executing an instruction out of flash. This coefficient represents the average energy consumption of an instruction executing out of flash (see Figure 5.11).
- E_{RAM} The average energy of an instruction which is executed out of RAM.

Developer specified

- R_{space} This parameter is the maximum amount of RAM (in bytes) that can be used for code. In cases where there is a limited amount of RAM, this constraint will have to be applied to ensure that everything fits into the RAM. Sometimes the amount of RAM can be determined statically, using heap and stack analysis [130].
- X_{limit} Some applications may have performance constraints. The maximum overhead allowed in the solution is constrained by this parameter — setting $X_{limit} = 1.1$ allows a 10% overhead in the number of cycles.

All of these parameters can be determined statically for the majority of deeply embedded platforms. The number of cycles per basic block, C_b, and iteration count of each basic block, F_b, are the only parameters which are sometimes challenging. In the majority of platforms, the cycle count is static and can be determined before runtime. The iteration count is more difficult since it can vary largely depending on the data an application is working on. However, the iteration count needs only to influence which basic blocks are likely to be hot and an estimate frequently works well — see page 74 for a discussion of estimating the number of times a basic block is executed, and the results section (Section 5.4.3) for its efficacy.

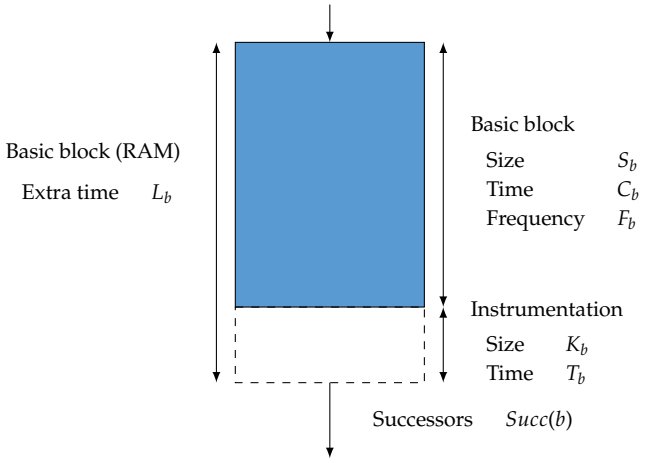


Figure 5.14: Parameters characterising each basic block.

Energy model

The energy consumption of a program can be estimated by forming a cost model. This cost model accounts for the number of cycles the processor spends executing out of RAM or flash, based on which basic blocks are placed in which memory, along with various other parameters describing the execution. Then, the sum of basic block energies can be minimised, by allowing an ILP (Integer Linear Programming) solver to select which basic blocks should be moved into RAM. Overall the problem is defined as the minimisation of the total energy of all basic blocks, by finding a set, R (used below), of basic blocks which are in RAM:

$$\text{minimise: } \sum_{b \in B} E(b). \tag{5.19}$$

The following equation defines $E(b)$, the energy of a basic block, b,

$$E(b) = (C_b + O_c(b) + O_r(b)) \cdot M(b) \cdot F_b, \tag{5.20}$$

where C_b is the estimated number of cycles taken to execute b, $O_c(b)$ is the overhead (in cycles), if present, from instrumenting the basic block with long range branches. The parameter $O_r(b)$ is a contention overhead, causing the basic block to take additional cycles when executing out of RAM. $M(b)$ returns the memory energy coefficient for the basic block and F_b is the execution frequency (iteration count) of the basic block. The memory energy coefficient, $M(b)$, gives the power per cycle when executing b,

$$M(b) = \begin{cases} E_{ram} & b \in R \\ E_{flash} & b \notin R, \end{cases} \tag{5.21}$$

where E_{ram} and E_{flash} are the RAM and flash energy costs respectively, and R is the set of basic blocks that are in RAM.

The cycle overhead of instrumenting a basic block is determined by considering the set of instrumented basic blocks, I,

$$O_c(b) = \begin{cases} T_b & b \in I \\ 0 & b \notin I, \end{cases} \tag{5.22}$$

where T_b is the cycle overhead for instrumenting basic block b, and I is the set of instrumented basic blocks. This set is constructed by considering where the successors of b are, in relation to b. The block only needs to be instrumented if one of its successors is in a different memory to itself.

$$\begin{array}{ll} b \notin I & \text{if } b \in R \text{ and } \forall (s \in Succ(b)) : s \in R \\ b \notin I & \text{if } b \notin R \text{ and } \forall (s \in Succ(b)) : s \notin R \\ b \in I & \text{otherwise,} \end{array} \tag{5.23}$$

where $Succ(b)$ returns the set of basic blocks that are immediate successors to b. This information is statically extracted from the control flow graph.

The RAM contention overhead parameter $O_r(b)$ is defined,

$$O_r(b) = \begin{cases} L_b & b \in R \\ 0 & b \notin R, \end{cases} \tag{5.24}$$

where L_b is the cycle overhead of placing a basic block in RAM. This stems from the problem that when instructions are executing from RAM and a load instruction is encountered, additional stall cycles may occur due to contention in accessing the RAM.

Constraints

Additional constraints need to be inserted into the ILP model to ensure that the resulting set of basic blocks is valid. This ensures that the total size of the code placed into RAM does not exceed available limits. The amount of code that can be placed into RAM is heavily dependent on the program. Some of the time static analysis can estimate the amount of free RAM, by considering the amount of variable storage and stack usage. However, in the case of complex heap usage, or dynamic stack usage it can be difficult to statically analyse, and therefore this parameter becomes a design point, needing to be specified by the developer (similar to stack and heap size).

Using R_{spare} as the amount of spare RAM that can be used for code, the following constraint ensures the generated solutions fit into the RAM,

$$\sum_{b \in R} (S_b + O_s(b)) \leq R_{spare}, \tag{5.25}$$

where S_b is the size of the basic block (in bytes), and $O_s(b)$ is the space overhead for when a basic block is instrumented, and in RAM. Similar to the Eq. 5.22, the size overhead of instrumentation is,

$$O_s(b) = \begin{cases} K_b & b \in I \\ 0 & b \notin I, \end{cases} \tag{5.26}$$

where K_b is the overhead cost in bytes to instrument the basic block.

An execution time constraint can also be formulated, allowing the execution time overhead to be restricted,

$$\sum_{b \in B} \Big((C_b + O_c(b) + O_r(b)) \cdot F_b \Big) \leq X_{limit} \cdot \sum_{b \in B} (C_b \cdot F_b), \tag{5.27}$$

where X_{limit} is the maximum factor the execution time can increase by. For example, if $X_{limit} = 1.2$, the solver allows the execution time to increase by up to 20%.

To be effectively minimised the constraints must be translated into inequalities, then solved by an Integer Linear Programming solver, such as GLPK [131].

Iteration estimation

A separate framework of constraints is used to make a crude estimate of the number of times each basic block is executed. The estimation requires information about the call graph of the program, the CFG, and the loops in the program, then returns an execution count for each basic block in the program.

A simple heuristic for the number of executions of a block is counting the absolute 'loop depth' of a block, d_b. This is found by performing a depth-first search on the call graph and CFG, counting the maximum number of loops the block appears under. For example, in the code snippet below, the statement `global += i;` is inside both the loop in `function2` and the loop inside `function1`, so has a depth, $d_b = 2$.

```
int function1()                          int function2()
{                                        {
    for(int i = 0; i < 10; ++i)              for(int i = 0; i < 10; ++i)
        function2();                             global += i;
}                                        }
```

The computed depth can be used with a loop trip count estimate, I, for an approximate number of executions of each block,

$$F_b \approx I^{d_b}, \tag{5.28}$$

for any $b \in B$. The estimation is simple, but provides enough of a heuristic to weight basic block frequencies for use in the model in some cases. However, the heuristic does not consider the control flow graph, and does not necessarily weight blocks inside the loop correctly. A more accurate estimation can be made by assigning this heuristic to just the loop headers, and then attempting to derive other block's iteration count based on the control flow. The estimation asserts that the sum of the edges entering the header block of the loop should be equal to the sum of the edges leaving the block,

$$E^{in}_{b_i} = \sum_{b_i \in Succ(b_j)} E_{j,i} \tag{5.29}$$

$$E^{out}_{b_i} = \sum_{b_j \in Succ(b_i)} E_{i,j}, \tag{5.30}$$

where $b_i, b_j \in B$, and $E_{i,j}$ represents the branch frequency from b_i to b_j. E^{in}_b and E^{out}_b represent the total of the edge frequencies entering and leaving block, b, respectively. While E^{in}_b and E^{out}_b should be equal, these are calculated separately because E^{in}_b or E^{out}_b should not be asserted if the block has no predecessors or successors, respectively. This case occurs in the first block of a function and blocks where the function returns,

$$F_b = E^{in}_b \qquad \text{iff. } b \text{ has a predecessor} \tag{5.31}$$

$$F_b = E^{out}_b \qquad \text{iff. } b \text{ has a successor.} \tag{5.32}$$

Finally, if the block is the header of a loop, the block iteration count, F_b, is at least equal to the previously mentioned simple heuristic,

$$F_b \geqslant I^{d_b} \qquad \text{iff. } b \text{ is a loop header node,} \tag{5.33}$$

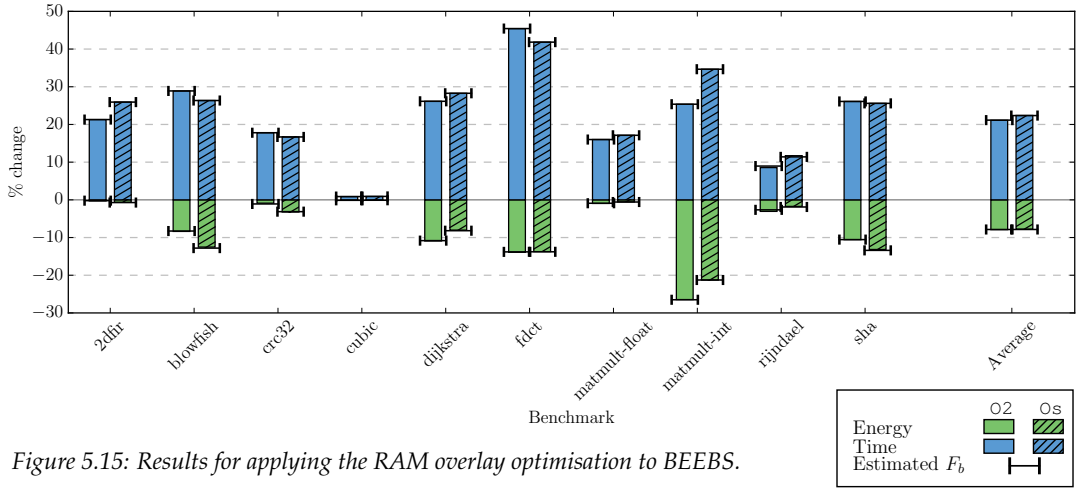


Figure 5.15: Results for applying the RAM overlay optimisation to BEEBS.

where d_b is a value calculated on the absolute loop depth of the block in the program.

These constraints are given to an ILP solver, which returns the estimate of basic block frequency. In some cases, the control flow graph results in a problem for which a canonical loop structure cannot be found. In these cases the irreducible control flow is ignored and the simple heuristic based on loop depth (mentioned earlier) is used.

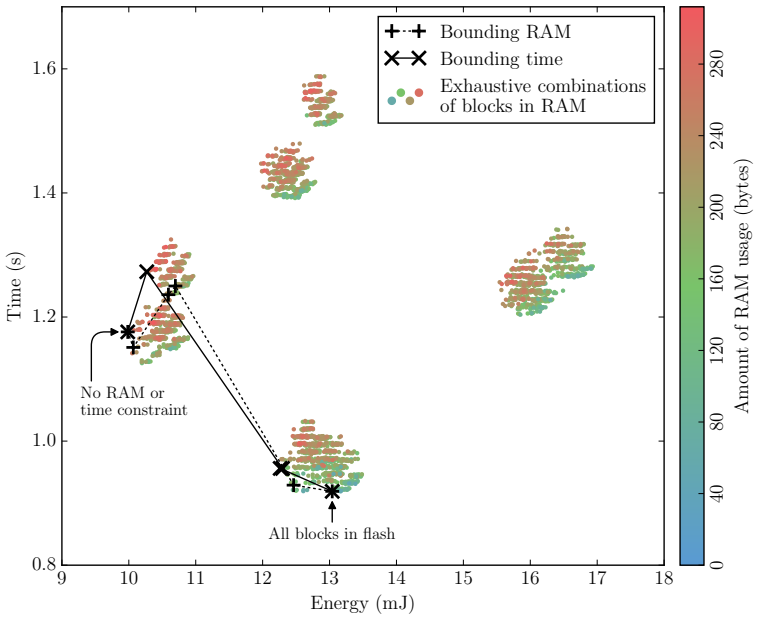
5.4.3. *Results*

The optimisation was evaluated on BEEBS at several different optimisation levels. These results are shown in Figure 5.15. Overall, the optimisation manages to reduce energy by an average of almost 10%, while increasing the execution time by 20%. Some benchmarks, such as *dijkstra*, *fdct* and *matmult-int* had significant reductions in energy consumption — up to 26%. Other benchmarks do not see the same benefit, with little reduction in energy consumption. Most of the difficulty in optimising these benchmarks is a result of the implementation of the optimisation. Many hot functions used in these benchmarks are library functions which are only available at link time and not to the compiler. In particular, the floating-point support for the STM32F1 is emulated with library calls by the compiler. This code is typically called very frequently and ideally would be placed into RAM for lower energy consumption. Fully integrating the optimisation into the linker would allow visibility of these functions.

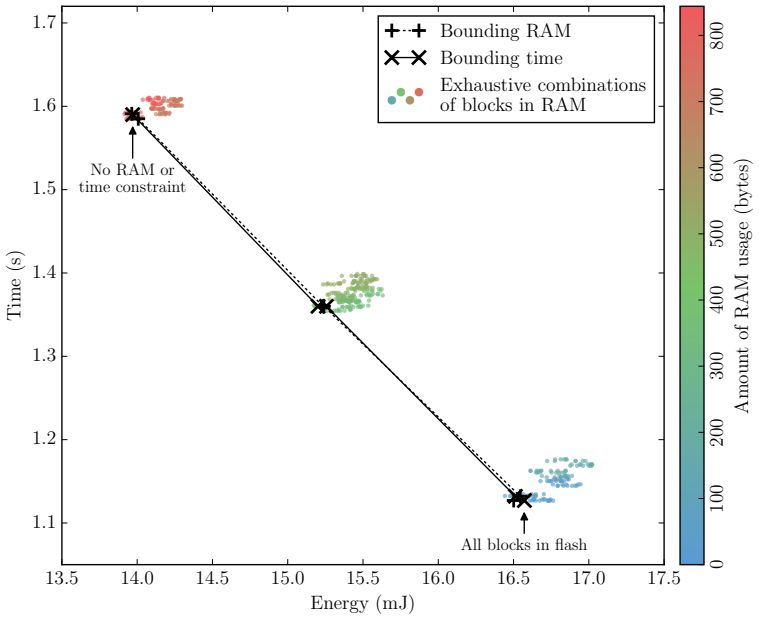
The graph also compares the results obtained by using the exact iteration count of a basic block against to the static estimate, shown as the symbol, $\mapsto$, on the graph. This indicates that similar solutions are picked the majority of time, even if the exact iteration count of the basic block is not known. In very few cases (*rijndael*) a slightly worse solution is picked (less than 1% worse), likely due to small inaccuracies in the model.

While the solution that is chosen by the solver is effective, and optimal in terms of the model, it may not be the global minimum for the energy consumption. To explore how good the chosen solution is in absolute terms, an exhaustive search of all possible configurations was carried out for two of the benchmarks. The results of this exploration are shown in Figures 5.16a and 5.16b, where each point is a possible combination of basic blocks in RAM or flash. The points are coloured by their usage of RAM, and the solutions selected by the solver are the lines on top of the points.

Both graphs have the majority of the points clustered around several regions, rather than



(a) matmult-int



(b) fdct

Figure 5.16: A plot of the execution time and energy consumption for all combinations of basic blocks in RAM for two benchmarks. The lines indicate solutions chosen by the solver as either the execution time overhead (X_{limit} increased) or the spare RAM bounds are relaxed (R_{spare} increased).

SoC	Align	Overlay	Comments
STM32F0	!	✓	The large 4-byte model coefficient suggests that the alignment optimisation may not be beneficial.
STM32F1	!	✓	The large 4-byte model coefficient suggests that the alignment optimisation may not be beneficial.
ATMEGA	✓	×	Small coefficients for the small boundaries suggests a similar optimisation may be beneficial. Separate program memory from data memory.
XMEGA	?	×	Unknown whether the alignment optimisation can be applied (requires profiling of the system). Separate program memory from data memory.
PIC32	✓	✓	Medium 4-byte coefficient but also large coefficients for the higher boundaries. Unified memory address space.
MSP430F	✓	✓	Medium 4-byte coefficient but also large coefficients for the higher boundaries. Unified memory address space.
MSP430FR	×	?	Flat energy profile for FRAM means alignment has little effect. Currently unknown whether FRAM is significantly lower power than RAM.
AM335x	×	×	Does not execute directly out of flash.
Epiphany	×	×	Does not execute directly out of flash.
XMOS	×	×	Does not execute directly out of flash.

Key	Applicable
✓	Yes
!	Possibly
×	No
?	Unknown

Table 5.6: Applicability of the energy optimisations to each SoC. Align is the code alignment in flash, and overlay is the RAM overlay optimisation.

uniformly spread across the plane. The clusters form due to particular basic blocks having a large influence on the solution. In the *fdct* case (Figure 5.16b), there are two similarly-sized large basic blocks which are executed frequently. Placing both in RAM gives the most energy efficient solution, the top left cluster. Placing either block in RAM results in the central clusters with a moderate increase in time and decrease in energy, and placing neither of the blocks in RAM forms the points in the bottom-right cluster. The same situation occurs for *matmult-int* — there are three important basic blocks, and eight clusters present in the graph (the lowest cluster is actually two overlapping).

On this graph, the optimal solutions for a combination of time and energy are the Pareto frontier — the points on the bottom-left edge of the graph. The solutions are close to these optimal solutions in most cases, and more importantly the solutions which are significantly worse in either energy consumption or execution time are avoided.

A near optimal solution is found, but the optimal is not because of simplifications in the model. In particular, the model assumes that none of the load instructions access flash memory which does not hold true for some code structures. Items such as constant data and initial values are often held in flash and will trigger higher energy consumption if blocks accessing these are placed in RAM.

Overall the optimisation is effective at saving energy (an average of 10%, up to 26% lower energy), with some limitations from the implementation — if fully integrated into the compiler, many of the benchmarks which did not have significant energy savings may have larger energy reductions.

5.5. Conclusion

This chapter developed two new energy optimisations, whose structure is fundamentally different from optimisations for performance. The optimisations exploit fundamental characteristics of how the hardware works — the layout of the flash memory and the energy efficient RAM memory — as well as taking account of the trade-off with longer execution time. Table 5.6 lists the applicability of the optimisations to each SoC used in this thesis, along with reasons why it is or is not applicable. Neither of the optimisations are suitable for the more powerful AM335x or Epiphany, neither of which use flash memory. Also, the XMOS device is not suitable, since at start-up the contents of an external flash chip are copied into RAM.

5.5.1. *Code alignment*

The first optimisation — aligning code based on a model of embedded flash — results in a small amount of energy saving in simulation, but the savings are very small when measured on the actual hardware. The model assigned energy values to each 2^k-byte boundary crossed by a sequence of flash accesses. The exact model parameters for the majority of SoCs mean that reducing the energy caused by embedded flash is often impossible. In particular, the transition cost associated with the 4-byte boundary is often high. The 4-byte boundary occurs because the flash is laid out into rows of 32 bit-lines, and accesses straddling the boundary will incur a large switching overhead. Crossing of the smaller boundaries occurs most frequently in the benchmarks — the majority of code executes sequentially without branching, and even when it does branch, the majority are local, crossing only small distances. These facts make it challenging to reduce the number of 4-byte transitions.

The total flash energy consumption for each benchmark is a large portion of the overall energy consumption — 19.5% of the STM32F1's energy consumption is attributable to address boundaries being crossed in the flash memory. In this case, the majority is caused by the 4-byte alignment parameter, accounting for 18.7% of the total energy consumption (i.e only 0.8% is caused by the other model parameters).

The optimisation may be more effective when the weighting of the parameters is skewed towards the higher boundaries. The ATMEGA SoC has model parameters which are low for the smaller boundaries, and high for the larger boundaries, such as changing a page. This allowed the optimisation to minimise the energy more effectively, however, the coefficients for this platform were still small overall, so only low amounts of energy could be saved. For this SoC, the cause of the energy consumption is more evenly distributed between its parameters. Due to the large number of SoCs using embedded flash and the large number of technology nodes, it is likely that there are many SoCs with a parameter profile suitable to this kind of optimisation.

It is possible that a completely different optimisation utilising the developed model could save energy. One such possibility is the reordering of basic blocks, rather than just a realignment. Reordering basic blocks allows much more flexibility in the selection of which boundaries a basic block crosses. Further opportunities for optimisation are possible, including splitting up basic blocks and rewriting code to change how it traverses memory and to thereby reduce energy consumption.

5.5.2. *RAM overlay*

The second optimisation developed exploited the difference in energy consumption between code executing out of flash, and out of RAM. On the STM32F1, code executing directly from

RAM consumes 35%–60% less energy, depending on the types of instructions executed. An optimisation can exploit these characteristics by placing the code to be executed into RAM, however, only a portion of the code can be moved to RAM — the flash memory is an order of magnitude bigger than RAM, plus the RAM is used for the stack and data as well.

The RAM overlay optimisation created a model of the whole program's energy consumption, based on the number of cycles spent executing out of either flash or RAM. The model can be minimised using integer linear programming, specifying which basic blocks should be moved into RAM to get minimal energy consumption, while keeping the amount of necessary space low and balancing the overheads of executing out of RAM. Overall, the energy consumption was reduced by up to 26% for some benchmarks, at the expense of also increasing execution time by up to 45%.

The increase in execution time was a result of the extra branching required to jump between the two memories, and extra latency incurred when executing out of RAM (contention between instruction fetch and load instructions). The first overhead could be eliminated by altering the memory map of the SoC, allowing the shorter range branch instructions to be used. The second overhead could be removed by adding an additional read port to the RAM, however, this may reduce the energy saving if it increases the power dissipation of the RAM.

The RAM overlay optimisation was evaluated on just the STM32F1 SoC, but can be applied to any deeply embedded SoC with flash that also has a unified address space. The Cortex-M series, MIPS and MSP430 processors all have a unified address space and often execute directly from flash; these processors would benefit from this energy optimisation. In particular the MSP430 has the address space of its RAM close to the flash, so could be implemented with low overhead.

5.5.3. *Energy effect and research questions*

Traditional instruction-level energy models would not have captured these effects, since the attributable energy consumption is orthogonal to the computational energy costs. Thus, both of the models found in this chapter could could be used in conjunction with an instruction-level energy model to enable better whole system modelling. The models can be used to further develop new optimisations which can target energy minimisation.

The research questions posed in Chapter 2 asked whether a class of optimisations exists which lower energy consumption by lowering average power (an optimisation's effect on k_P, the scaling of power), rather than decreasing the execution time (the effect on k_T, time scaling). Both optimisations attempt this, with the RAM overlay being most successful. As an example, the best result for this optimisation is when it is applied to *matmult-int*, at the `O2` optimisation level, resulting in an energy reduction of 26%, even in the presence of a 24% execution time increase. This translates to the optimisation having $k_T = 1.24$, and $k_P = 0.59$ — a significant reduction in the average power dissipation of the SoC and as mentioned previously, these factors result in an energy reduction since,

$$k_T \cdot k_P < 1 \tag{5.34}$$

$$1.25 \cdot 0.59 < 1 \tag{5.35}$$

$$0.74 < 1. \tag{5.36}$$

The optimisation being able to reduce significant amounts of energy by reducing the k_P coefficient suggests that this optimisation belongs to a class of optimisations which can lower the energy consumption in this way. Both of the optimisations in this chapter differ significantly from typical execution time optimisations — both are applied to the whole program at the end of compilation, and consider where the source of the instruction is rather than what computation

they perform. In this case, the optimisations consider the code alignment, and which memory in which the instructions are stored.

The approach used in this chapter to generate new optimisations for energy can be repeated. In general the approach is as follows:

1. Identify unusual energy behaviour through empirical testing and examination of the target SoC. The features identified could be low-level activity, such as transistor switching activity, or the operation of specific functional units. For example, identifying that absolute address alignment impacts energy consumption when executing directly from flash.

2. Construct a model which allows this behaviour to be investigated from a higher level. The model relates some aspect of program execution to its energy behaviour. For example, the embedded flash energy model relates an instruction stream to the sequence of memory accesses, and that sequence of memory accesses to an energy consumption figure.

3. Use the model in the compiler. The model can be used to construct an optimisation by informing decisions about when and where it is most beneficial to make code modifications. This part is dependent on the exact characteristics being modelled. For example, the embedded flash model can be used to predict how a basic block should be aligned to minimise its flash energy consumption.

The procedure should be applicable across a variety of processors and to illustrate this a hypothetical application of this procedure follows. If a processor has a Direct Memory Access (DMA) controller, there may be cases when it is beneficial (for energy) to use this as opposed to typical memory copies. The first step of the procedure would analyse the DMA's energy behaviour and determine under what conditions the DMA performed favourably. A model could then be constructed, e.g. using the total amount of memory to be copied. Finally, an optimisation utilising the model may identify regions of code where it is beneficial to use the DMA instead of traditional instructions, such as stack initialisation, memory copies, and function epilogues and prologues.

Overall, this chapter has followed this methodology for two identified energy characteristics, developing models and then optimisations. The optimisation for reducing embedded flash energy was ineffective, however, there are still potential reductions for more complex optimisations. The RAM overlay optimisation managed to significantly reduce the energy consumption, purely by reducing the average power during execution, showing that compiler-based optimisations that are *designed for energy* can have a significant effect on energy consumption.

Chapter 6.
Combining optimisations

Optimisations often have significant and complex interactions, which are difficult to model. Therefore, empirical testing of combinations of optimisations is necessary to identify where the interactions occur. This is even more important with optimisations for energy consumption, which may rely on subtle and fragile behaviours in the target platform. One example of this is the alignment optimisation seen in the previous chapter — an optimisation applied after aligning a code fragment may have a chaotic effect, causing the code to be misaligned, and have higher energy consumption.

The interactions between energy optimisations and execution-time optimisations have not been explored previously — it may be challenging to ascertain the effectiveness of an energy optimisation when used in conjunction with other optimisations. This chapter attempts to answer research questions about the combinations of energy optimisations and execution time optimisations. In particular it is important to know whether there are significant interactions between them. If there are interactions, it may not be possible to simply enable energy optimisations on top of the best optimisation level a compiler offers — a new set of optimisations may have to be found. To explore the these interactions, the RAM overlay optimisation developed in the previous chapter is used.

Firstly, the efficacy of individual optimisations for time is compared by reusing the fractional factorial design technique, finding optimisations whose effectiveness increases or decreases when used in combination with the energy optimisation. Then, the best optimisation sets, as found by a genetic algorithm are tested with the energy optimisations — testing whether the energy optimisation could be simply enabled on top of existing optimisations. Finally, the genetic algorithm is rerun, allowing it to select whether to apply the RAM overlay optimisation, as well as the execution time optimisations. This explores whether possible interactions could enable even lower energy consumption.

6.1. Fractional factorial design

Section 4.4.1 of Chapter 4 explored the existing optimisations in the compiler using a fractional factorial design design methodology. This is repeated here, except with the RAM overlay optimisations applied as well as the existing compiler optimisations. The reader is referred back to Section 4.4.1 for an explanation of fractional factorial design.

By rerunning the same design with the RAM overlay optimisation applied, the interactions between the existing optimisations for execution time and the new energy optimisation can be found. This allows optimisations whose efficacy changes with the new optimisation to be identified. It is expected that optimisations which affect the control flow graph structure and basic block size will affect the RAM overlay optimisations — the size of the blocks has a key role in choosing which code goes into RAM. Also, code which changes the number of loads and stores in a block may affect the optimisation, since this changes the additional overhead when a block is placed in RAM (contention on the bus).

6.1.1. *Results*

The majority of optimisations do not have their efficacy significantly changed when the RAM overlay optimisation is applied. A good example of this is shown in Figure 6.1, which shows the

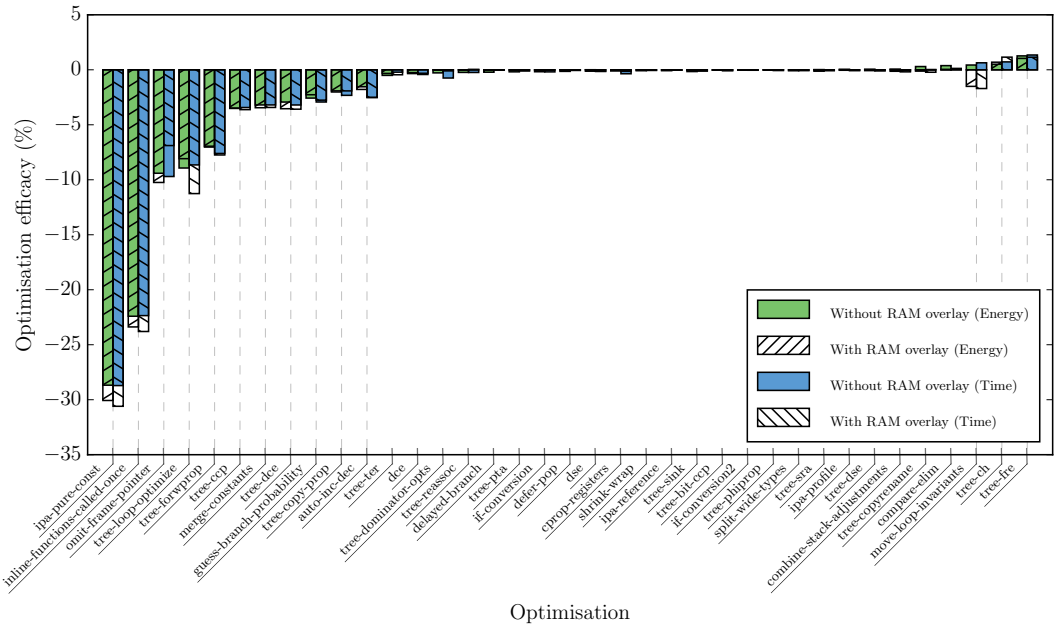


Figure 6.1: Comparison of effective optimisations for blowfish (`O1`), on STM32F1, with and without the RAM overlay optimisation applied.

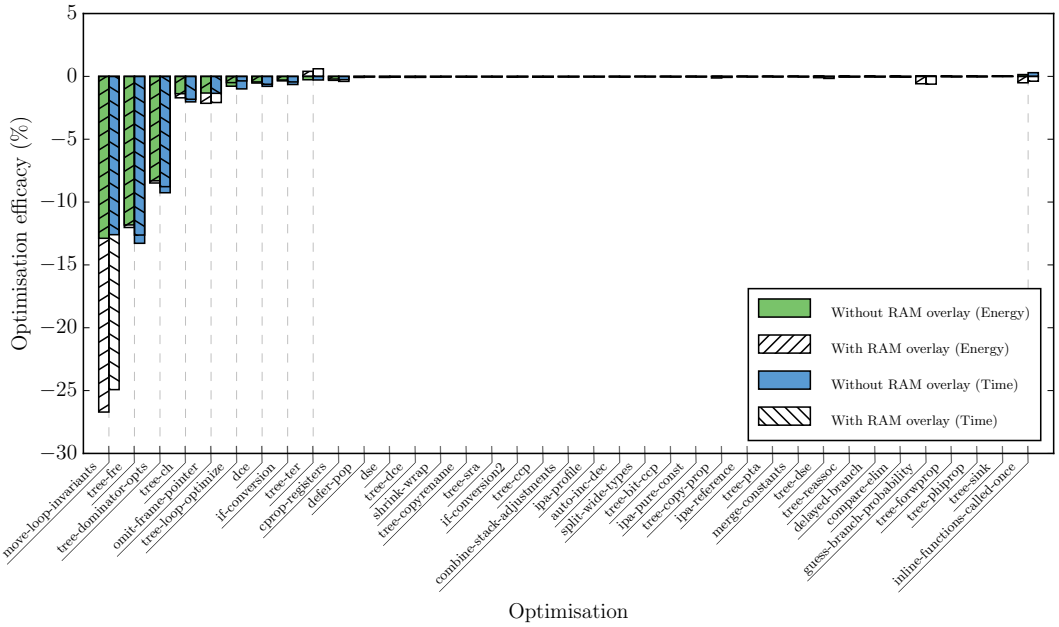


Figure 6.2: Comparison of effective optimisations for dijkstra (`O1`), on STM32F1, with and without the RAM overlay optimisation applied.

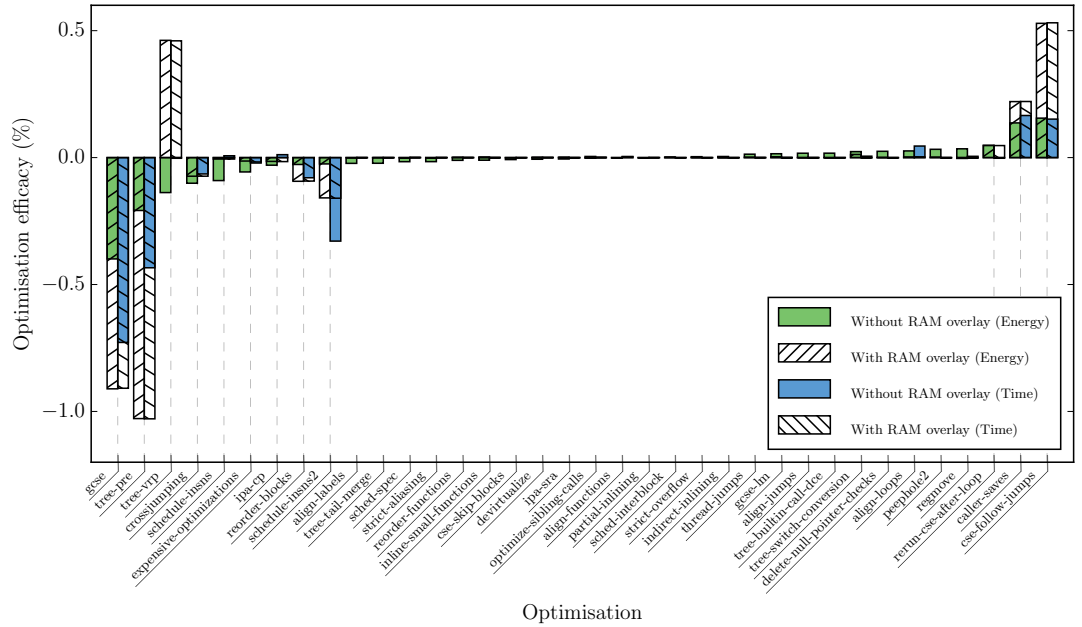


Figure 6.3: Comparison of effective optimisations for 2dfir (`O2`), on STM32F1, with and without the RAM overlay optimisation applied.

fractional factorial design results for both with and without the RAM overlay optimisation (the latter being identical to the computed results in Chapter 4). The green bars represent the effect an optimisation has on energy, while the blue bars represent its effect on time. The overlaid, hatched versions of each bar are the results with the RAM overlay optimisation.

The second graph, Figure 6.2, shows the *dijkstra* benchmark. The results are similar to that of the previous graph (Figure 6.1) in that energy and time both decrease together, however there is an optimisation which performs differently when the RAM overlay optimisation is applied. This optimisation is `move-loop-invariants`, attempting to move code out of a loop which does not need to be computed each iteration (see page 103). The effect of this optimisation must be indirect — the `move-loop-invariants` is applied before the RAM overlay optimisation, therefore it affects how the overlay is applied. The efficacy is achieved because moving code out of a loop will decrease both its size and execution time, making it more likely to be placed into RAM by the overlay optimisation. The optimisation appears as effective for execution time too, since moving code out of a loop which is placed in RAM will also decrease the overhead of that block being in RAM (fewer loads causing contention on the bus).

The final graph, Figure 6.3 (*2dfir*) has an optimisation which does not have a significant effect when the RAM overlay optimisation is not applied, but has a negative effect when applied. The optimisation (`tree-vrp`) performs "value range propagation", a type of analysis pass that attempts to propagate interval information about registers and variables to other instructions. This can enable the deletion of null pointer checks, modifying the control flow graph. In turn, this leads to larger basic blocks (no splitting where pointer checks would occur), making it more difficult for the basic block to be moved into RAM.

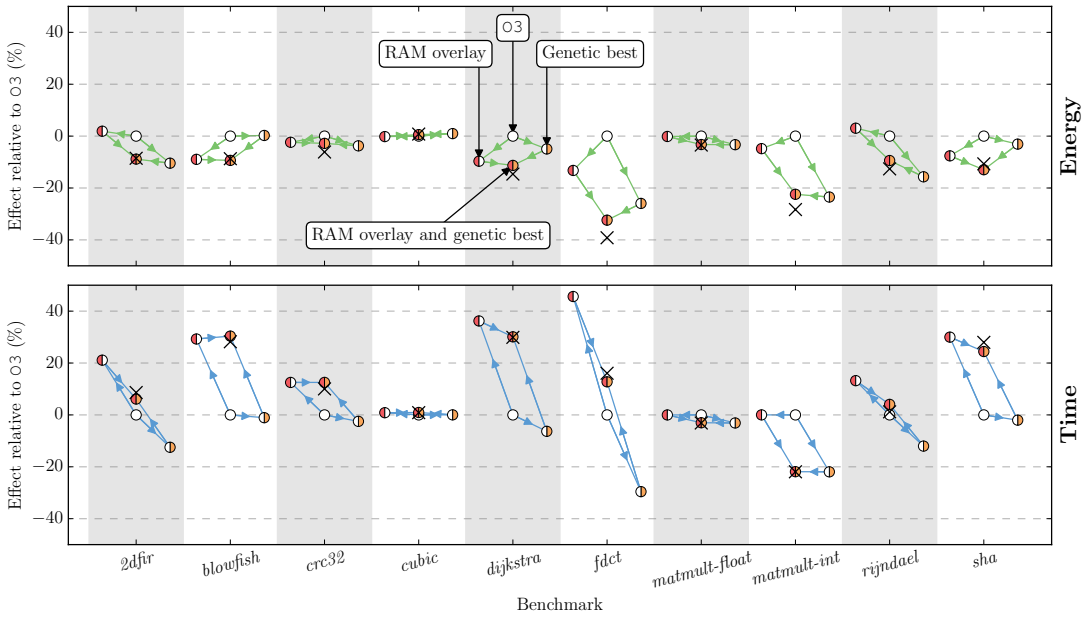


Figure 6.4: A plot showing the effect of all combinations of applying the RAM overlay, and the set of optimisations found by the genetic algorithm, compared to `O3`. A positive percentage indicates an increase in energy consumption or execution time. A negative percentage indicates a decrease.

6.2. Known good sets

Chapter 4 used genetic algorithms to find a good set of optimisations for each benchmark. The set of optimisations resulted in up to 27% decrease in energy and 30% decrease in execution time, compared to `O3` on the STM32F1 platform. These sets of optimisations are found using just the existing optimisations in the compiler. By applying an energy optimisation on top of this good set of optimisations, it can be seen whether the optimisation still reduces energy, answering one of the research questions proposed in Chapter 2 — whether a new choice of optimisations would be needed to combine with the energy optimisations.

Figure 6.4 plots the effect on energy and time of applying the best optimisation set found (genetic algorithm, optimising for energy), the RAM overlay optimisation, and both the best optimisation set and the RAM overlay together. These are represented by the points on the graph, with the arrows indicating the application of the optimisation. When these points form a parallelogram the optimisations combine linearly — there are no significant interactions between the set of optimisations chosen by the genetic algorithm, and the RAM overlay optimisation. The cross marker for each benchmark indicates where the combination of the best optimisation set and RAM overlay would be if the combination was linear. For example, the energy consumption for *sha* when the RAM overlay is applied is the left-most point under *sha*, at an 8% reduction

Benchmark	Energy (%)	Time (%)
2dfir	-3.2	-18.4
blowfish	-2.4	-4.9
crc32	-1.5	0.0
cubic	-1.3	0.0
dijkstra	-1.6	-0.2
fdct	-1.1	-0.1
matmult-float	-1.7	-0.1
matmult-int	-0.5	3.0
rijndael	-7.7	-7.2
sha	-0.8	-4.6
Average	-2.2	-3.3

Table 6.1: The additional effect on time and energy when using a genetic algorithm and applying the RAM overlay optimisation together.

relative to `O3`. The energy consumption when the best optimisations found by the genetic algorithm are applied is the right-most point, reducing the energy by 3%. The expected energy reduction when both are applied is 11%, marked by the cross. However, the actual reduction in energy consumption is slightly larger, at 13%.

The majority of the results suggest that optimisations compose well together — there are only a few cases where the actual energy is significantly different to the estimation via summing the effects. In the graph, this manifests as most of the points forming a parallelogram shape. The result is promising, suggesting that this energy optimisation is mostly orthogonal to the other performance optimisations applied. Overall, up to 32% of the energy consumed at the best compiler optimisation level can be saved.

Three of the results have slight deviations from linear composability — *dijkstra*, *fdct* and *matmult-int*. For these benchmarks there it is likely that the structure of the basic blocks has changed slightly, resulting fewer blocks that can be placed in RAM.

The second part of the figure shows the execution time corresponding to the energy consumption. These figures are much as expected: the RAM overlay optimisation increases execution time, and the genetic-algorithm-found optimisations reduce the execution time (since energy and time are mostly proportional). There is also more composability of energy optimisations and time optimisations when examining the execution time — this is expected, because as an optimisation metric, energy consumption depends on execution time, *as well as* other factors.

6.3. Genetic algorithms

The previous section tested whether the best optimisation set found for the existing compiler optimisations composed linearly with an energy optimisations, however did not ascertain whether this set could be improved upon. Here, the genetic algorithm is rerun, with the energy optimisation as an extension to the gene, allowing it to be turned on or off. By analysing the resultant sets for differences, the question of whether a different set of optimisations is needed to enable an effective energy optimisations can be answered.

The genetic algorithm used is identical to the one in Section 4.5.1, with the addition of an extra bit in the gene to specific whether the RAM overlay should be applied or not.

Overall the optimisation sets produced are very similar to the set without the energy optimisation, and the resulting energy is not significantly different. This indicates there are few

interactions between the existing optimisations and the RAM overlay optimisations.

Table 6.1 show the additional benefit that can be achieved by the genetic algorithm with the RAM overlay optimisation as part of the gene. The genetic algorithm always chose to enable the RAM overlay, and there is very little improvement that can be achieved in terms of energy for most of the benchmarks — shown by an average of 2.2% improvement in energy. The *rijndael* benchmark, however, does see an improvement of 7.7%. The benchmark performed poorly when just the RAM overlay optimisation was applied on top of `O3` because it contains a large number of big basic blocks. The genetic algorithm which included the RAM overlay optimisation allows a different set of optimisations to be found, and this optimisation set allowed the energy optimisation to be effective in this case.

There is little extra reduction available in execution time too, as shown by the time column in the same table. As with energy, the execution time for *rijndael* is reduced, however, there is also a large reduction in the execution time of the *2dfir* benchmark. Upon examination of the individual optimisations selected by the algorithm, the set changes and disables the `tree-vrp` optimisation. As identified in the previous section (Section 6.1.1) this optimisation was one of the few whose effect changed from positive to negative when applying the RAM overlay optimisation. The genetic algorithm exploited this, disabling the optimisation to slightly reduce the energy consumption and greatly further reduce the execution time.

6.4. Conclusion

In Chapter 2, the effect an optimisation had on energy, time and power was given,

$$T'_a = k_T \cdot T_a \tag{6.1}$$

$$P'_a = k_P \cdot P_a \tag{6.2}$$

$$E'_a = (k_P \cdot P_a) \times (k_T \cdot T_a), \tag{6.3}$$

where E_a, T_a, and P_a, are the energy, time and average power before the optimisation's application, respectively. These are transformed by the optimisation's effect on time, k_T, and power, k_P, to form E'_a, T'_a, and P'_a. One of the questions posed asked whether k_P and k_T affected each other when two different optimisation sets were applied: one set reducing k_T (execution time optimisations), and one set reducing k_P (energy optimisations). This chapter sought to understand the interactions, and its effect on the choice of execution time optimisations.

Few significant interactions were found between the optimisations for time and the RAM overlay optimisation, analysed using fractional factorial design. The fractional factorial design allowed the change in efficacy to be measured for individual optimisations, and only one optimisation, when considered across the whole benchmark suite, was found to change from having a positive impact to a negative impact (`tree-vrp`).

The fractional factorial design analysis examined individual optimisations, but did not look at many optimisations combining together. Genetic algorithms were used in Chapter 4 to combine optimisations together and find an improvement in energy over the `O3` optimisation level. Using the per-benchmark sets of optimisations found, the RAM overlay optimisation was applied, examining whether there were significant increases or decreases in both the energy and time. In most cases the composition was linear: there were no significant interactions between the best set of time optimisations and the energy optimisation. For this particular energy optimisation its effect on the energy and execution time was mostly orthogonal to the other optimisations. This is because the operation performed is mostly independent of patterns in the code's structure and the exact nature of the calculation. In general, energy optimisations which follow this pattern

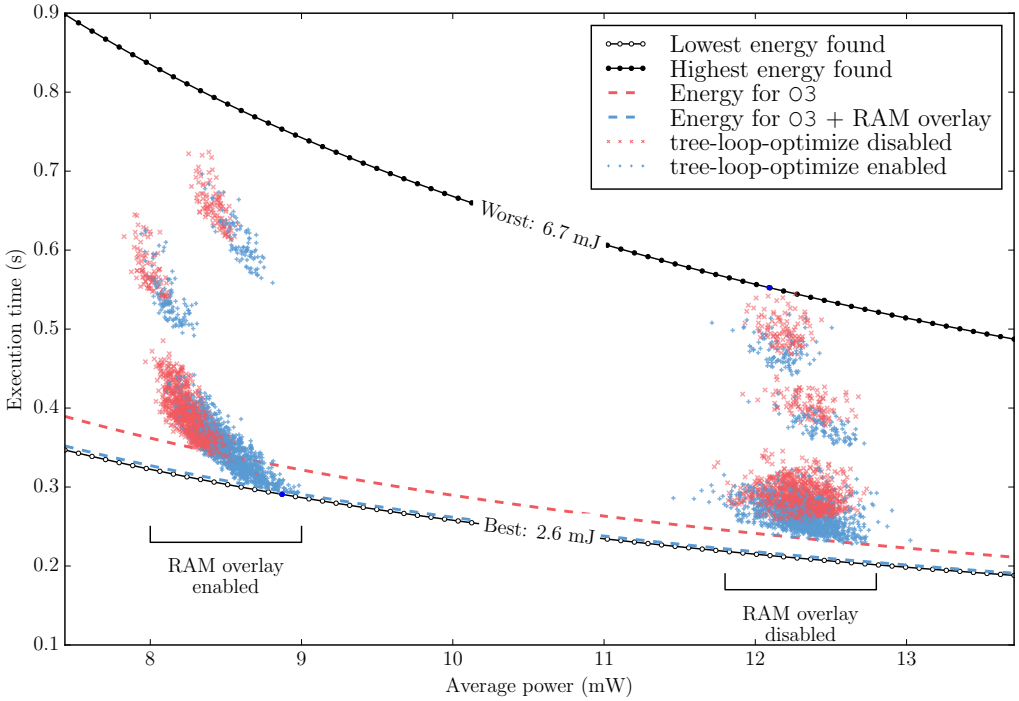


Figure 6.5: A scatter plot of power and time values. Each point is a different combination of compiler optimisations. The clusters on the left have the RAM overlay optimisation enabled, whereas the clusters on the right do not have the RAM overlay applied.

should always be orthogonal, and combine linearly with existing optimisations. An additional optimisation set on which to apply the RAM overlay was found not to be needed — applying the RAM overlay onto both `O3` and the genetic best set of optimisations resulted in close to linear composability in most benchmarks, for energy consumption and execution time. The two sets complemented each other well — the RAM overlay reduced energy consumption at the expense of execution time, whereas the execution time optimisation set reduced both metrics. When using the genetic algorithm best set, the average execution time increase dropped from 17.9% increase (just RAM overlay) to 8.5% (both), as well as improving the energy reduction, from −4.3% over `O3` to −11.7%.

One benchmark, *rijndael*, did not benefit from the RAM overlay optimisation at all, however, when the genetic algorithm was rerun with the energy optimisation, an alternate set of optimisation was found that enabled energy savings. The new version of the genetic algorithm was identical, but allowed the energy optimisation to be applied. With the energy optimisation applied, the algorithm could search for a set of optimisations which enabled the energy optimisation, resulting in a 7% reduction in energy (for *rijndael*). There were minimal decreases in energy for the other benchmarks — again suggesting few interactions and linear composability.

While this analysis cannot account for all energy optimisations — only one was tested in combination here — it adds weight to energy optimisations being a distinct class of optimisation which can be applied orthogonally to time optimisations.

Figure 6.5 shows many combinations of optimisations for the *blowfish* benchmark on STM32F1. Each point is a different combination of optimisations comparing the average power to the exe-

cution time. On this graph, an optimisation which generally causes a decrease in execution time will shift the points down on the y-axis, whereas an optimisation which causes a change in the average power causes a shift on the x-axis. This is shown by highlighting the `tree-loop-optimize` optimisation in red and green — the main shift is vertical. When highlighting the points for all other time optimisations, the points only shift along the time axis, with no significant effects along the power axis. This adds weight to the argument that none of the existing optimisations significantly affect energy via power reduction. On the other hand, the RAM overlay optimisation does affect the power — the clusters on the left side of the graph have the RAM overlay enabled, whereas the points on the right have it disabled.

Overall the existing optimisations in the compiler do not seem to significantly affect the efficacy of the RAM overlay energy optimisation. The energy optimisation can be applied on top of a set of existing optimisations, without decreasing the efficacy of the existing optimisations, although the reduction in energy and time provided by the energy optimisation can be variable (since it is dependent on the sizes of the basic blocks). This has implications for the design of energy optimisation in general — specific energy-reducing features in the compiler can be focused on separately from other optimisations, without having to account for interactions between them. For the application developers this is also beneficial, since the energy optimisation can be composed easily with typical optimisations that are already applied.

Chapter 7. Conclusion

A compiler optimisation will affect key metrics of the program under test, such as energy, time and average power. As first discussed in Chapter 2, the energy, E_a, of the program, a, can be calculated from the total time taken, T_a, and the average power, P_a,

$$E_a = P_a \times T_a. \tag{7.1}$$

When an optimisation is applied, all of these metrics change too. The average power is scaled by the coefficient, k_P, and the execution time is scaled by k_T,

$$T'_a = k_T \cdot T_a \tag{7.2}$$
$$P'_a = k_P \cdot P_a \tag{7.3}$$
$$E'_a = (k_P \cdot P_a) \times (k_T \cdot T_a), \tag{7.4}$$

where E'_a, T'_a, and P'_a are the energy, time, and power, respectively, after the optimisation has been applied. This equation shows that the resulting energy consumption after a transformation is dependent both on the execution time and the average power. Thus, a reduction in energy can be achieved by either lowering the average power or the execution time. A reduction in energy can be achieved even in the event of the optimisation increasing the time or power. Overall, a reduction in energy is possible if $k_P \cdot k_T < 1$.

In Chapter 2 it was hypothesised that the majority of pre-existing optimisations in compilers achieve a lower energy consumption primarily by reducing the k_T coefficient. Chapter 3 developed a benchmark suite, with characteristics suitable for testing compiler optimisations and energy consumption. Chapter 4 examined the hypothesis that existing optimisations are designed purely for time, finding that for almost all existing optimisations the energy efficacy was a result of a lower k_T coefficient. The k_P parameter does not stay constant, however, varying slightly for each optimisation. This is purely a side effect of the optimisation not explicitly considering power at all — the effect on power is incidental.

Chapter 5 demonstrated another class of optimisations exists, which lowers the energy by explicitly lowering the average power and achieving a lower k_P. These optimisations focus on lowering the average power of the program, even in the case where this meant an increase in execution time. The RAM overlay optimisation was successful in reducing energy consumption, by up to 26% and reducing average power by up to 41%.

When combined with the traditional optimisations in the compiler (Chapter 6), the RAM overlay proves to be largely independent, achieving a similar reduction in energy consumption even with different sets of optimisations also transforming the code. This suggests that the energy optimisation can be applied independently in most cases, and will combine with other methods to efficiently select compiler optimisations.

7.1. Existing compiler optimisations

"Do existing compiler optimisations save energy purely by reducing the k_T coefficient?"

"Are there instances of standard compiler optimisations which affect k_P?"

The standard compiler optimisations in GCC were examined extensively, analysing their effect on energy and time, in groups (optimisation levels), individually (using fractional factorial design) and in carefully chosen combinations attempted to find the best possible set. Overall the majority of the optimisations affect energy by modifying the k_T coefficient. A few optimisations also affect k_P: the optimisation flags `schedule-insns` and `schedule-insns2` both influence k_P in different ways. These optimisations perform instruction scheduling before and after register allocation, respectively (see Section 4.4.3 for more detail).

The analysis performed on individual optimisations was used to determine whether a single group of optimisations is effective for multiple benchmarks or platforms. Overall, there are optimisations which are effective for the same benchmark over different platforms, and the same platform for different benchmarks. However, there is not a set of optimisations, or a single optimisation which is consistently effective for all benchmarks and platforms.

Since there is a lack of optimisations which affect the change in power without also affecting the execution time, this suggests there may be a new class of optimisations which specifically target average power reduction. It is not unexpected that the existing optimisations have a much larger effect on execution time than energy consumption, since the original design goals of these optimisations is to make the code faster to execute.

Overall this means that there is only marginal additional benefit to targeting energy consumption over execution time when purely utilising existing compiler optimisations — in almost all cases the execution time can be used as a proxy for energy consumption. Since energy consumption is more challenging to measure than execution time, an effective way to optimise for energy in absence of measurement equipment is purely to minimise execution time. This is shown by the genetic algorithm finding similar solutions when optimising for energy and for time. For all benchmarks the energy and time savings achieved are very similar, with a maximum of 5% difference when choosing to minimise energy.

7.2. Optimisations for energy

"Is there a class of optimisations which lower energy consumption via reducing k_P?"

"Are these optimisations structured or applied differently to regular optimisations?"

"Can these optimisations effectively reduce energy?"

Two energy characteristics were explored, with the objective of exploiting these features and developing an optimisation to reduce energy via reducing the average power. Both of these characteristics involve the embedded flash memory present in the majority of the deeply embedded SoCs, and can be exploited since the code is often executed directly from the flash. The first effect is the structure of embedded flash affecting how much energy is necessary to load from a specific location, and when executing code its absolute position can cause different amounts of energy to be consumed. Secondly, accessing flash is more power hungry than accessing RAM, and code can be moved from flash to RAM to lower the overall energy (lowering energy by up to 26%, with an average reduction of 10% in our experiments).

These optimisations are of a fundamentally different structure to that of execution time optimisations. While optimisations for time typically reduce and reorder the code, these optimisations tend to focus on how and where the code is executed. The difference in structure suggests that optimisations which target energy consumption must be structured differently to typical compiler optimisations. Another difference between these optimisations is that the energy optimisations take a global view of the code, and are performed late in the compilation process. The optimisations both require very low-level information about the SoC, mapping this

to the higher compiler level. This is akin to pipeline modelling for performance optimisations, but requires much more hardware specific detail.

This changes the way that the search for optimisations is conducted. For many compiler optimisations the effect is local, and based on restructuring a small amount of code. For effective energy optimisations a global view of the program must be taken, and the entire context of the execution should be taken into account (i.e. the hardware, and its environment). These points are reflected in other optimisations targeting energy — DVFS needs to have a view of the whole program, as well as the possibly accessing peripherals to change the voltage and the frequency [103]. Inserting sleep modes [4] also requires this level of global information, and specific details about the processor's `idle` instruction.

Section 5.5.3 suggested the following methodology for finding new optimisations to reduce energy:

1. **Identify unusual energy behaviour through empirical testing and examination of the target SoC.** Controllable hardware features which affect the energy consumption must be found.

2. **Construct a model which allows this behaviour to be investigated from a higher level.** A model is necessary to relate the high-level operation of a stream of instructions, or other software features to the energy consumption of the hardware.

3. **Use the model in the compiler.** The model can be used to inform compiler optimisations, determining what modifications should be made to the code.

The most challenging step of this procedure is identifying unusual energy behaviour, since often this requires extensive testing and energy measurement hardware.

7.2.1. *Code alignment*

The energy consumption of a section of code that is executed directly from flash depends on its absolute position in the memory. Effects such as crossing a page boundary increase the energy cost of a sequence of accesses to the memory. An energy model of the embedded flash memory was constructed, allowing the flash memory's energy to be calculated for an instruction sequence. An optimisation was developed to utilise this model by realigning basic blocks so that fewer costly boundaries are crossed.

The optimisation is shown to reduce the energy, however the reduction is small. In some platforms, such as the STM32F1, it is not possible to significantly reduce the embedded flash's energy, since the largest cost is crossing a 4-byte address boundary. These crossings happen frequently and many are necessary, resulting in only a minimal possible energy reduction. Other platforms, such as ATMEGA, have small coefficients for the small boundaries. These can be more effectively optimised, however for this particular platform the flash memory represented a small fraction of the total SoC energy.

Overall, the optimisation resulted in very little energy being saved — in many of the SoCs a large fraction of the flash energy is an unavoidable cost. However, there is possibly the scope for a more complex optimisation to utilise the embedded flash energy model to make code placement decisions.

7.2.2. *RAM overlay*

The RAM overlay optimisation exploited the difference in energy consumption when executing directly from RAM and directly from flash. In general, an instruction consumes 30–60% less energy when executed from RAM instead of flash. When combined with the fact that a small

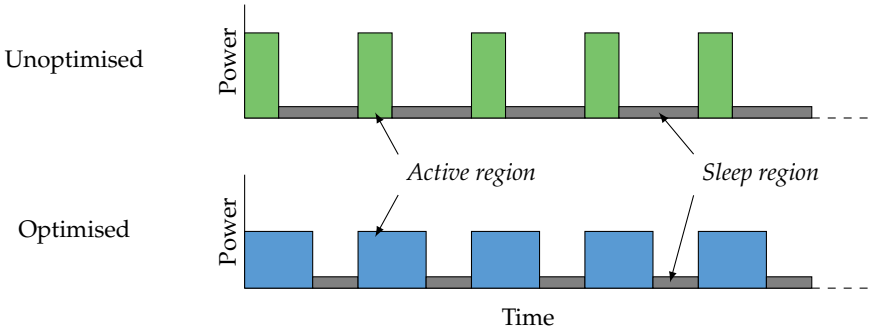


Figure 7.1: A periodic application before and after the application of the RAM overlay optimisation.

portion of the code is the majority of the runtime, the most intensive regions of code can have their energy minimised while using only a small amount of RAM.

The optimisation identifies the hot regions of code and places them into RAM, while balancing the overhead from doing this. The main overheads stem from the memory regions for RAM and flash being in different parts of the address space and requiring indirect branching to jump between them. A model is constructed to account for this effect, and this model is minimised, giving a set of basic blocks that should be placed in RAM.

The optimisation saves up to 26% of the application's energy, only requiring a small amount of RAM. However, the optimisation also increases the execution time by up to 40% at the same time. While the increase in execution time can be problematic for some applications, it also means that the average power is significantly lower — 41% lower in some cases. For periodic applications (see Figure 7.1) this can be beneficial — the significantly lower power and increased execution time results in a lower overall energy for these applications. This occurs even in the case where the RAM overlay was unable to decrease energy for the active region, but still increased execution time and reduced average power — Figure 7.2 exemplifies this case and how the energy is lowered through a reduction in the sleep time.

The optimisation is an effective way to save energy, and can be applied to applications which periodically wake to perform computation with successful results. The specific implementation of the algorithm, as a post-compiler pass, limits its efficacy. However, if it was fully integrated into the linker more effective decisions could be made, putting frequently executed library code into RAM and saving energy on the benchmarks for which are currently beyond the optimisation's reach (benchmarks using emulated floating-point and extensive library calls).

7.3. Combining optimisations for time and optimisations for energy

"Do optimisations designed to lower k_P affect optimisations which lower k_T?"

"Are there significant interactions between the two classes of optimisation?"

"Is there a different 'best' set of optimisations when including energy optimisations?"

The combination of the RAM overlay optimisation and the existing optimisations in the compiler gives insight into how an energy optimisation affects optimisations for performance. The RAM overlay optimisation is mostly orthogonal to the existing optimisations — the effects on both time and energy compose linearly when the two are combined. This suggests that these types of energy optimisations are independent from execution time optimisations, with the interaction

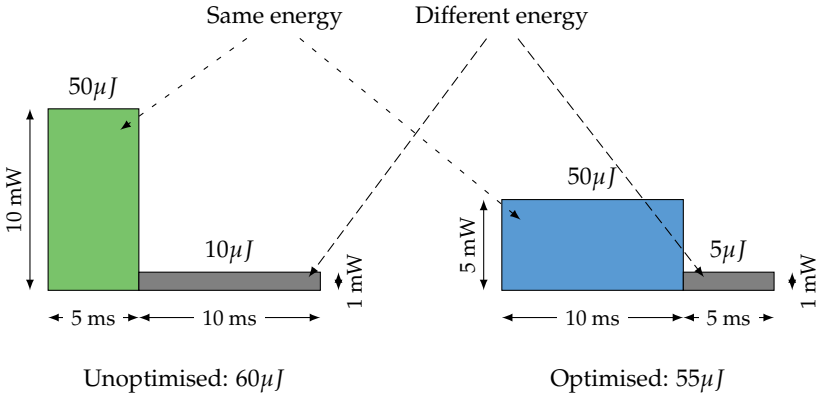


Figure 7.2: Even if the RAM overlay does not reduce energy for the active region, in a periodic application the reduced power and increased time leads to an overall lower energy.

causing only 1–2% change in energy or time. This conclusion is not necessarily intuitive — the RAM overlay occurs after all other optimisations have been applied and thus should be affected by the changes to the code. Particularly for the RAM overlay optimisation, the lack of interactions with other optimisation is due to its global nature, and it attempting to always put a maximal amount of code in RAM. If the code of one block changes and becomes unsuitable for RAM placement, a different block or combination of blocks will be selected which has the next best energy consumption.

The fractional factorial design exploration of individual optimisations was repeated with the RAM overlay enabled. Only one case was found where an optimisation had a significantly different effect when the RAM overlay was applied (`tree-vrp`), going from a beneficial effect on energy to an increase in it. This optimisation propagated information through the IR, informing other optimisations about the ranges of values variables can take, allowing them to be more effective. When this optimisation was combined with the RAM overlay, it affected the code structure in such a way that fewer basic blocks could achieve a lower energy. All other optimisations did not significantly change with the RAM overlay in efficacy — they either increased or decreased the energy to a similar degree.

A genetic algorithm could not find a set of optimisation which performed significantly better when also given the RAM overlay optimisation to enable or disable, and in all cases chose to enable the RAM overlay. This further suggests that the RAM overlay is mostly independent of existing optimisations.

7.4. Future work

There is much future work to be done in all of the areas this thesis covers, including compiler optimisation, energy efficiency and embedded systems. This section discusses some of the individual extensions that could be made to the issues explored in this thesis, and then larger overall goals.

7.4.1. *Further research questions*

While this thesis extensively explored the research questions posed, there are many additional questions raised for the specific topics within. In particular, the two potential optimisations

presented in Chapter 5 require a global view of the program — the deficiencies in the RAM overlay optimisation are mostly caused by the optimisation not having full visibility of the code within the library functions. Other existing 'energy optimisations', such as DVFS also use the whole program to make decisions, and it is currently unknown whether an energy optimisation generally requires this global view to be effective.

Since the proposed code-alignment optimisation was only marginally effective, an optimisation which utilises the model in a more novel way may be able to achieve larger savings. An optimisation such as reordering basic blocks, or even radically restructuring the code so it does not cross expensive boundaries may be possible. The model itself can be extended, based on the observation that the flash wait-states may affect the coefficients of the model, as well as DVFS which may require different energy coefficients for each voltage-frequency pair. Further exploring the structure of flash to find additional effects may also be fruitful, such as whether the exact value stored by the flash affects its energy. Previous work in this area by Joo et al. suggests that in some cases this could be significant [132].

The RAM overlay optimisation could also benefit from exploring the trade-offs with flash wait-states and DVFS. In particular, with flash wait-states the RAM becomes faster to execute from than the flash. While this could be handled already by the model in Section 5.4.2, some minor modifications may need to be made. The model can also be easily extended to the case where the basic blocks must be transferred to RAM at the beginning of execution. This needs to be taken into account in the case where the SoC goes into a very deep sleep that loses all RAM contents, and the code must be reloaded on waking. Larger modifications are necessary if the RAM overlay is extended to a fully dynamic approach, loading and evicting code as necessary [5], although the benefits of this approach for deeply embedded systems are not immediately obvious.

The chapter on combining energy optimisations with time optimisations (Chapter 6) only looked at a single optimisation — this could be extended to look at combinations of multiple different optimisations for energy consumption, and whether they also combine linearly. This first requires additional effective optimisations for energy to be devised.

7.4.2. *Future research direction*

The existence of a class of optimisations specifically for energy consumption draws parallels with the autovectorisation attempts currently in compilers — both exploit features that can be specific to individual SoCs and attempt to change the code in a specific way to achieve these goals (by utilising specialised hardware). When more optimisations for energy are added to this class, they will likely only apply to specific hardware, and possibly require machine-assisted or developer tuning (as with the amount of available space in RAM for code, with the RAM overlay).

A significant question is whether these energy optimisations will still be effective when used in a multi-threaded setting. In some cases, multi-threading makes the modelling phase of constructing an optimisation more challenging. For example, many instruction-level energy models propose using the number of bit flips between consecutive instructions as part of the model [12]. This approach encounters difficulties with Simultaneous MultiThreading (SMT) type processors (for example, the XMOS processor), where consecutive instructions may be from separate threads. The interleaving of these threads can change frequently, particularly when data dependent effects change the control flow, causing the exact sequence of instructions down the pipeline to be unknown [13]. When caches are also involved with SMT, different methods of partitioning problems across the threads have differing performance [133], and likely differing energy effects.

Compiler optimisations can also be effective in a multi-*core* setting where each core can be

put to sleep while waiting on other cores to compute data. In this case, an energy optimisation should attempt to maximise the number of cores that are in a low-power state, and the length of time they are in that state. This could involve moving certain items of code between threads, or even moving threads between cores with different performance and efficiency trade-offs, as in ARM's big.LITTLE [134]. Managing the complexity of these problems is an ongoing research topic. Overall, it is possible that there are even larger energy savings to be made when considering parallel programs.

The majority of this thesis has focused on embedded systems, and it is currently unknown whether the answers to the research hypotheses hold for larger, more powerful systems, such as desktop computers and servers. These systems all utilise formidable memory hierarchies, which add non-determinism and make the efficacy of an optimisation more challenging to estimate — it is more dependent on the context of the program's execution.

As more optimisations and optimisations of high complexity are added to the compiler, more advanced methods are needed to select and order these transformations to ensure maximum efficiency. The inherent problem behind this is the unpredictability of an optimisation's effect and unclear relationships between the code before and after the transformation. The RAM overlay optimisation exhibits this behaviour — the ILP solver can choose a completely different set of blocks to be placed in RAM with just a small tweak of the code. This presents a trade-off between achieving maximal effectiveness, and predictability of the optimisation — if a greedy approach of placing the most frequently executed basic block in RAM had been taken, the optimisation would be more predictable, but would not have decreased the energy as much. There is evidence that using features of the code's structure and machine learning can be used to guide how and when to apply an optimisation, and it is possible that this could similarly be used to guide the application of energy optimisations.

Overall the compiler's efficacy is limited by the layers of abstraction between the software and the hardware. The use of very high-level languages further detaches the actual energy consumption of the hardware from the structure of the code that is running. Often this makes it challenging to optimise the code in a way that will reduce energy consumption.

Compilers and related tools have typically been one-way translation methods from these programming languages to very low-level assembly code. To effectively optimise the reverse must happen — low level energy behaviour and artefacts must be translated back upwards to the software and the associated toolchain. Such mappings have been extensively explored between the hardware and instructions in the form of instruction-level energy models (and the flash-memory model in this thesis), but there are few methods of reliably mapping up to the programmer's level from assembly code. For compilers to be more effective at energy reduction, development of these tools is required.

The lack of transparency from the compiler level down to the hardware's energy-consumption level leads to compilers currently being limited in optimisation capability. With current compilers, only a small energy saving can be achieved, whereas with tools which use low-level energy behaviour at the application level much larger savings may be possible.

This page is intentionally blank.

Appendices

This page is intentionally blank.

Appendix A.
Optimisation Reference

This appendix lists some of the standard optimisations commonly mentioned in this thesis, and describes how they work and their potential benefits.

Branch chaining. Branch chaining is an optimisation which reduces the amount of branching that needs to be performed [100]. After other optimisations have been applied, the code may result in a branch which targets another branch. The optimisation replaces the initial branch's destination with the final destination, skipping the intermediary branch instructions. This reduces the amount of branching, making the program execute faster.

```
    ; miscellaneous computation          ; miscellaneous computation
    b label                              b next
    ...                                  ...
label:                          ⇒    label:
    b next                               b next

next:                                next:
    ...                                  ...
```

The above example has two branches. The first branch jumps to `label`, which just executes another branch to `next`. The optimisation replaces the first branch with a branch to `next`.

Common subexpression elimination. Common subexpression elimination removes expressions which appear multiple times in the code, storing the result and using this stored result in place of repeating the computation [135]. This reduces the number of instructions that must be executed. The optimisation can be applied whenever an expression appears identically on the same control flow path.

```
a = (b + 1) * (b + 1);      ⇒      tmp = b + 1;
                                   a = tmp * tmp;
```

In the above example, the expression `b + 1` appears twice, so is stored in the temporary, `tmp`, in the transformed code.

Constant folding. Constant folding is an optimisation that identifies expressions which result in a constant, by evaluating the operations performed on them [135]. For example, an add operation with two constants as operands can be collapsed to a single constant. Folding the constants reduces the number of instructions which need to be executed at runtime.

```
z = 10 + 12;                ⇒      z = 22;
```

The above example folds the `10 + 12` expression into `22`. While a programmer is not likely to initially write such an expression, the constant values could be the result of another optimisation.

Constant propagation. Constant propagation allows the compiler to propagation knowledge about which variables have constant values in them [135]. This can enable further optimisations, such as constant folding, as well as reducing the amount of runtime work that needs

to be performed. Often constants can be embedded directly into the instruction, rather than loading a value (as would be done with a variable), giving additional speed gains. To perform the optimisation, the compiler must be certain that the value in the variable is constant.

```
x = 10;                          x = 10;
y = 12;                    ⇒     y = 12;
z = x + y;                       z = 10 + 12;
```

In the above example, the compiler knows that the values of `x` and `y` are constant, and therefore can replace the uses of the variable.

Copy propagation. This optimisation is very similar to constant propagation, except propagates variables which are equal to another variable [135]. When the compiler can determine the value stored in that variable is identical to another variable, either variable can be chosen. This can reduce the number of registers needed, and the amount of load and stores necessary to retrieve variable contents.

```
x = a;                           x = a;
y = x + 10;                ⇒     y = a + 10;
```

The above example propagates the assignment of `a` to `x` into the last expression.

Dead code elimination. Dead code elimination removes code from the program which is never executed [135]. This often occurs after other optimisations are applied and can result in reducing the total amount of computation that is performed. The code can also reduce the amount of branching, if a particular branch of a condition is shown never to be executed.

```
int function(int x)              int function(int x)
{                                {
    return x * 2;          ⇒         return x * 2;
    int y = x + 1;
}                                }
```

In the above example, the assignment of the variable `y` is not reachable and therefore the expression computing this value can be removed without affecting the functionality of the program.

Expression simplification. This optimisation is the application of simple algebraic rules to simplify any mathematical operations the code performs [135]. These operations often reduce the total number of instructions required to compute the result. For example applying distributivity rules, $a \times (b + c) \equiv a \times b + a \times c$, can simplify the expression $a \times b + a \times c$ into $a \times (b + c)$. The latter expression requires only one multiplication instead of two. However, the trade-off can be complex, since the longer expression has fewer interdependencies and could possibly take advantage of instruction level parallelism. Also, the optimisation can only be applied to certain types — operations such as floating point often cannot be rearranged this way.

```
x = a * 1;                 ⇒     x = a;
```

The example above applies the identity, $x \times 1 \equiv x$, to the code allowing the expression to be simplified.

Function inlining. The function inlining optimisation reduces the overhead of calling functions, by copying the body of the function into the caller [136]. While this does increase code size, it allows further optimisations to be applied to the combined code, and removes branching and stack modification overhead. The compiler must determine how many times the function is called, and how large the function is to decide whether it is likely to be beneficial to inline the function.

```
int fn1(int y)                          int fn1(int y)
{                                       {
    return y * y;                           return y * y;
}                                       }
                                  ⇒
int fn2(int x)                          int fn2(int x)
{                                       {
    return fn1(x) + 1;                      return x * x + 1;
}                                       }
```

The above example inlines the function `fn1`, into `fn2`, removing the overhead of calling `fn1` in `fn2`. The original function will remain in the code if it is called elsewhere or not deemed beneficial to inline.

Reorder blocks. This optimisation involves using knowledge about the likely result of a conditional branch instruction, enabling blocks to be reordered, and fewer jumps taken. This should result in fewer execution cycles, since taking a conditional branch typically requires more cycles to execute than not taking the branch. The optimisation tries to place the most likely block directly after the branch in memory, allowing the faster 'fall-through' path to be taken.

```
if(x == 0)
    y++;
else
    y = y * z;
```

```
    ; unlikely to be true                 ; likely to fallthrough
    cmp r0, #0                            cmp r0, #0
    bne else_condition                    beq if_condition

    mul r1, r2                  ⇒         add r1, #1
    b end                                 b end
else_condition:                       if_condition:
    add r1, #1                            mul r1, r2
end:                                  end:
```

In the above example, an analysis pass has determined that the condition `r0 == 0` is unlikely to happen, and therefore inverts the condition

If-conversion. If-conversion attempts to remove the branches created by conditional statements in the code [137]. It does this by attempting to create an equivalent branch-free section of code. If-conversion is only typically beneficial for short conditional blocks, or on processors with very large branching penalties.

```
if(x > 10)
    y = 0;                    ⇒     y = y & -(x > 10);
```

The example above transforms the `if` statement into a simple assignment expression. The expression exploits the combination of arithmetic, bitwise and conditional operators — the conditional operator in the parenthesis evaluated to either one or zero. If it is zero, the entire expression becomes `y` bitwise-anded with `0`, resulting in zero. Otherwise, the one is negated and bitwise-anded with `y`. This preserves the value in `y`, since `-1` is represented by every bit set.

Instruction scheduling. Instruction scheduling is an assembly-level backend optimisation [135]. The optimisation attempts to reorder instructions, to reduce stalls in the processor's pipeline. Many processors do not have full instruction bypassing implemented in the pipeline, therefore an instruction which requires the result of an immediate predecessor may be stalled, while waiting for the instruction's results to be written into the register file. The reordering attempts to move other instructions in the place of these stall cycles, maintaining the computational throughput.

```
add r0, r1                    add r0, r1
mul r0, r0          ⇒         sub r2, r3
sub r2, r3                    mul r0, r0
bic r3, r2                    bic r3, r2
```

In the above example, the instructions are interleaved, so that the `r0` and `r2` registers do not need to be read immediately after they are written to.

Jump threading. Jump threading is an optimisation which attempts to reduce the number of conditionals and branches that must be executed [138]. This occurs when a previous expression sets a variable in such a way that ensures a subsequent expression is always true or false. When this is the case, the control flow can jump directly from the first block into the subsequent block, bypassing the conditional and jump instructions. This reduces the amount of branching that must be performed and should increase the performance of the program.

```
if(x == 0)                    if(x == 0)
{                             {
    y = 0;                        y = 0;
}                                 goto label;
if(y == 0)          ⇒         }
{                             if(y == 0)
    a += 5;                   {
}                             label:
                                  a += 5;
                              }
```

In the above example, when `x == 0`, the following condition (`y == 0`) is always true. Therefore the execution can jump directly out of that `if` block into the second condition's block.

Loop fusion. The loop fusion optimisation takes two loops and attempts to combine them into a single loop [135]. This is most easily done when the number of iterations the loop performs is the same. By reducing multiple loops into one loop, the overhead is reduced. Other effects must be considered, since if the loops are large, the resultant loop may not fit into instruction cache and thus execute slower.

```
for(i = 0; i < 10; ++i)
    x++;

for(i = 0; i < 10; ++i)
    y = y * 2;
```

⇒

```
for(i = 0; i < 10; ++i)
{
    x++;
    y = y * 2;
}
```

The above two loops both have the same iteration count, and can be merged into a single loop.

Loop header copying. Loop header copying is an optimisation which duplicates the portion of the loop which checks whether the loop conditions still hold, exiting if they do not [139]. This optimisation can save some branching, and enables further optimisation of the loop — since the condition is duplicated at the end of the loop body, it can be optimised and scheduled with that basic block.

```
    ...
    ; loop entry
    b loop_header
loop_body:
    add r0, #1

loop_header:
    cmp r0, r1
    beq loop_exit
    bne loop_body

loop_exit:
```

⇒

```
    cmp r0, r1
    beq loop_exit
3   bne loop_body
loop_body:
    add r0, #1
    ; a new basic block here
    ; is not necessary now
    cmp r0, r1
9   beq loop_exit
    bne loop_body

loop_exit:
```

In the above example, the loop header (the compare instruction and two branches) is duplicated at the entry to the loop. Note that further optimisations would likely remove the branches on lines 3 and 9, since these fall through into the following code.

Loop interchange. Loop interchange (also called loop permutation) is an optimisation which permutes the order of a nested loop structure, based on some criteria [135]. These criteria take into account the sequence of memory accesses in the loop body, and how the access pattern would be changed by modifying the order of the loops. Often the loops are permuted so that data which is accessed in subsequent iterations is likely to still be in cache, providing a speed up.

```
for(j = 0; j < 1000; ++j)
    for(i = 0; i < 1000; ++i)
        a[i][j]++;
```

⇒

```
for(i = 0; i < 1000; ++i)
    for(j = 0; j < 1000; ++j)
        a[i][j]++;
```

In the above example, the loops are swapped in order. This may provide a speed boost due to the way the two-dimensional array is accessed in the loop body.

Loop invariant motion. Loop invariant motion is a loop optimisation which moves code out of a loop that can be shown not to change from iteration to iteration, i.e. it is invariant. This reduces the amount of redundant computation that must be performed, since the result of the expression is identical every time, and can just be computed once.

```
for(i = 0; i < 1000; ++i)
{
    a = x + y + z;
    array[i] = a * i;
}
```

⇒

```
a = x + y + z;
for(i = 0; i < 1000; ++i)
{
    array[i] = a * i;
}
```

In the above example, the expression, `a = x + y + z`, is moved out of the loop, since it does not depend on any expression derived from a loop variable.

Loop tiling. Loop tiling (also called loop blocking) restructures large loops by adding an extra loop, with a counter incrementing by the tile size [136]. The inner loop is then modified to iterate up to the tile size. This divides the larger loop into blocks, often enhancing the cache locality properties of the loop. The restructuring is most effective on multiple nested loops, where data is accessed locally, i.e. adjacent positions in a multidimensional matrix.

```
for(i = 0; i < 1000; ++i)
    for(j = 0; j < 1000; ++j)
        a[i][j]++;
```

⇒

```
for(ib = 0; ib < 1000; ib += 50)
  for(jb = 0; jb < 1000; jb += 50)
    for(i = ib; i < ib + 50; ++i)
      for(j = jb; j < jb + 50; ++j)
        a[i][j]++;
```

The example above shows two loops which are both tiled to sizes of 50 iterations. A further operation, such as loop interchange could swap the second and third two loops, resulting in the array being iterated over in tiles of 50 by 50. If the inner of the loop performed lots of local accesses (i.e. to adjacent array cells), then this keeps the relevant data in cache.

Loop unrolling. The loop unrolling optimisation attempts to minimise the loop overhead by expanding several iterations of the loop into a straight line [136]. This is most easily performed when the iteration count is known, and is a multiple of the unroll factor (otherwise a possibly conditional loop prologue or epilogue must be added to account for the extra iterations). The unrolling will also enable further optimisations to transform the code, providing additional gains. Loop unrolling can increase code size by a large amount, and care must be taken to not negate any of the optimisation benefits by creating code which does not fit in the instruction cache.

```
for(i = 0; i < 60; ++i)
    x = x + 1;
```

⇒

```
for(i = 0; i < 60; ++i)
{
    x = x + 1;
    ++i;
    x = x + 1;
    ++i;
    x = x + 1;
}
```

In the above example, the loop is unrolled three times, duplicating both the body of the loop and the loop increment expression (`++i`).

Loop unswitching. Loop unswitching is a transformation that moves conditional statements out of the inner body of the loop, duplicating the loop in the process [136]. This can only be performed if the result of the conditional statement is invariant for the duration of the loop. This can decrease execution time, at the expense of increasing code size, since there are fewer branches inside the loop, and possibly fewer instructions executed overall.

```
for(i = 0; i < 10; ++i)
{
    if(flag)
        b[i] = a[i];
    a[i]++;
}
```

⇒

```
if(flag)
    for(i = 0; i < 10; ++i)
    {
        b[i] = a[i];
        a[i]++;
    }
else
    for(i = 0; i < 10; ++i)
        a[i]++;
```

The above example shows a condition on `flag` inside the loop. This test is moved outside, and the loop duplicated.

Predictive commoning. Predictive commoning is an optimisation which attempts to reuse expressions that are computed inside loop bodies [140]. This optimisation is similar to common subexpression elimination, however, accounts for the recurrences of complex expressions inside loops. The optimisation minimises the number of array elements and expressions which must be loaded each loop iteration.

```
for(i = 0; i < 10; ++i)
{
    a[i+2] = 3 * a[i+1] + a[i];
}
```

⇒

```
t1 = a[0];
t2 = a[1];
for(i = 0; i < 10; ++i)
{
    t3 = 3 * t2 + t1;
    a[i+2] = t3;

    t1 = t2;
    t2 = t3;
}
```

In the above example, the first loop performs two loads each iteration. However, these loads are simply retrieving data previously stored by `a[i+2]`. In the transformation, the computed values are stored in `t1` and `t2` between loop iterations to minimise the number of array loads necessary.

Omit frame pointer. The frame pointer is used to mark the beginning of a function's frame on the stack. This is particularly useful for debugging and stack unwinding. However, it requires the use of an additional register and stack location for each function that is called. The omit frame pointer optimisations does not store this frame pointer, which can make debugging more difficult, but allows the register to be used for other variables. This will reduce spilling onto the stack in regions of high register pressure and decreasing execution time. Additionally, since the frame pointer is not saved, this reduces the required stack memory for a program, and removes load and store operations to the stack for this pointer, also reducing program execution time.

```
function:
    str fp, [sp, #-4]!
    mul r3, r0
    mov r0, r3
    ldr fp, [sp], #4
    bx  lr
```

⇒

```
function:

    mul r3, r0
    mov r0, r3

    bx  lr
```

The above example shows the removal of load and store operations to the stack where the frame pointer is saved. Also since the `fp` register is no longer used, this can be used for general purpose computation.

Redundancy elimination. Redundancy elimination is similar to common subexpression elimination, focusing on eliminating expressions which appear in multiple basic blocks in the control flow graph [136]. For example, if two identical expressions appear in all the predecessors to a block, the expression can be moved into the final basic block, rather than being computed in every block. This does not necessarily make the program execute faster, but should reduce the code size, which may increase speed, through instruction cache effects.

```
if(x < 50)                          if(x < 50)
{                                   {
    x++;                                x++;
    y = y * 2;                      }
}                                   else
else                          ⇒     {
{                                       x--;
    x--;                            }
    y = y * 2;
}                                   y = y * 2;
```

The above example has the expression `y = y * 2` in both paths through the code, so this can be moved above or below the conditional.

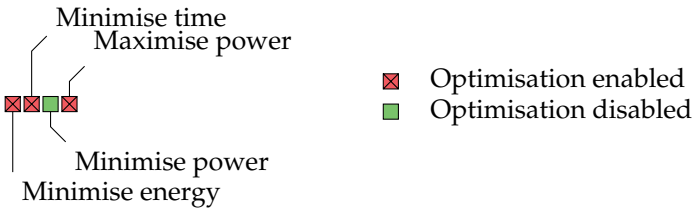
Strength reduction. Strength reduction is an optimisation that attempts to replace expensive operations with functionally equivalent and cheaper operations [136]. The classical example of this is multiplication by two, which can be replaced by adding the number to itself. This may speed up the program on some architectures where a multiply is more expensive than an addition.

```
x = x * 2;          ⇒     x = x + x;
```

In the above example, the multiplication is changed to the equivalent addition.

Appendix B.
Datasets

The following table lists the optimisation sets found for each benchmark on the STM32F1 SoC. Each benchmark has four different sets of optimisations — corresponding to the goals minimise energy, minimise time, minimise power, and maximise power.



The above diagram indiciates that the specified optimisation should be enabled to minimise power, but disabled for the minimise energy, minimise time and maximise power goals.

Optimisation	2dfir	blowfish	crc32	cubic	dijkstra	fdct	matmult-float	matmult-int	rijndael	sha
align-functions	⊠⊠□□	⊠⊠⊠⊠	⊠⊠⊠□	□□⊠□	⊠⊠⊠⊠	⊠⊠⊠□	□⊠□⊠	□□□□	⊠⊠⊠⊠	□□□□
align-jumps	⊠⊠⊠□	□⊠□⊠	⊠□□□	⊠⊠⊠⊠	⊠⊠□□	□⊠□⊠	□□□⊠	□⊠□⊠	⊠⊠⊠⊠	□⊠□⊠
align-labels	⊠□⊠□	⊠□□⊠	⊠□⊠□	□□□□	⊠□⊠□	□□□□	□□□□	□⊠□⊠	⊠□⊠□	□⊠⊠⊠
align-loops	⊠⊠□□	⊠□□⊠	⊠⊠⊠⊠	⊠□□□	□⊠⊠□	⊠□□⊠	⊠□⊠□	□□□□	□□□⊠	⊠⊠□⊠
auto-inc-dec	□□□⊠	□□⊠□	□⊠⊠□	⊠⊠□□	□□□□	⊠⊠□□	⊠□⊠□	□□□□	□□⊠□	□□⊠□
caller-saves	⊠⊠⊠⊠	□□⊠□	□⊠□□	⊠□□□	□□⊠⊠	□□□□	⊠⊠⊠□	⊠⊠⊠□	□⊠⊠□	□⊠⊠⊠
compare-elim	⊠□□□	⊠□□□	⊠⊠⊠□	□□⊠□	⊠⊠□⊠	⊠□□□	□□□⊠	□□□□	⊠⊠⊠□	⊠□⊠⊠
cprop-registers	⊠□□⊠	□⊠□⊠	⊠⊠⊠□	⊠⊠⊠□	□⊠□□	□⊠□⊠	⊠□□⊠	□□□⊠	□⊠□⊠	□□⊠□
crossjumping	□□□⊠	□⊠□⊠	□□□□	⊠□⊠□	□□□□	⊠□□⊠	□□□⊠	□⊠□⊠	□⊠⊠⊠	□⊠⊠⊠
cse-follow-jumps	⊠□⊠□	□□□□	⊠□□□	⊠⊠□□	⊠□⊠□	⊠□⊠□	□⊠□□	⊠⊠⊠⊠	⊠⊠□□	□□⊠⊠
cse-skip-blocks	⊠□⊠□	⊠□□⊠	⊠□⊠□	□□□⊠	□⊠⊠□	□□□⊠	⊠□□⊠	□⊠⊠□	□□□□	⊠⊠□□
dce	□□□⊠	□⊠□□	□□□□	□□□⊠	□⊠⊠⊠	□□⊠□	□⊠□□	□⊠⊠□	⊠⊠□⊠	□□□⊠
defer-pop	□□□⊠	⊠□□□	□⊠⊠⊠	□⊠□⊠	□□□□	⊠□□⊠	□□⊠□	□⊠□□	⊠⊠□⊠	□□⊠□
delayed-branch	□⊠□□	⊠⊠⊠□	□□⊠⊠	⊠⊠□□	⊠⊠□□	⊠⊠⊠□	⊠⊠□□	⊠⊠□□	□⊠□⊠	□□⊠⊠
delete-null-pointer-checks	⊠□□⊠	□□□□	⊠□□□	□⊠⊠⊠	□□⊠□	□□⊠□	□□□□	⊠□⊠□	□□⊠⊠	□□⊠□
devirtualize	⊠□⊠□	□⊠□⊠	⊠□⊠□	□⊠□□	⊠□⊠□	⊠□⊠□	□□□□	□□⊠⊠	⊠⊠□□	□□□⊠
dse	⊠⊠□□	□⊠□□	□□⊠□	⊠⊠□□	□□□□	⊠□□□	□□⊠⊠	⊠⊠□⊠	□□□□	□□⊠⊠
expensive-optimizations	⊠□⊠□	□□⊠⊠	⊠□⊠□	□□□⊠	□□□□	□⊠⊠⊠	□□□□	□□□□	□□□□	□⊠⊠⊠
gcse	□□□□	□⊠□⊠	□□□□	□□□⊠	□□□⊠	□□⊠□	□⊠⊠⊠	□□□⊠	□□□⊠	⊠□□⊠
gcse-after-reload	□□□□	□⊠⊠□	□□⊠⊠	□□□⊠	□□⊠⊠	⊠⊠⊠⊠	□□□⊠	□□□⊠	□□⊠□	⊠□□⊠
gcse-lm	□⊠□□	□⊠□⊠	□⊠□□	□□□⊠	⊠⊠□□	⊠⊠□□	□□□⊠	□□□⊠	□□⊠□	⊠□□⊠
guess-branch-probability	□⊠⊠⊠	⊠□⊠⊠	□⊠□⊠	⊠□□□	□□□□	□⊠□⊠	⊠⊠□⊠	⊠⊠□□	□⊠□□	□□□□
hoist-adjacent-loads	□⊠□□	□□□□	⊠□□□	□⊠□⊠	□⊠⊠⊠	⊠⊠⊠⊠	⊠⊠⊠⊠	⊠⊠⊠□	⊠⊠□□	□⊠□⊠
if-conversion	□□□⊠	□□⊠□	⊠⊠⊠□	□□□⊠	□□⊠□	□□□□	□⊠⊠⊠	□□□⊠	□⊠□□	□□□□
if-conversion2	□⊠□□	⊠⊠⊠⊠	⊠□□⊠	⊠□□□	□□⊠⊠	⊠⊠⊠□	□□□□	□□□□	□□⊠□	□⊠□⊠
indirect-inlining	□⊠⊠⊠	□□□⊠	□⊠⊠⊠	□□□□	□□□□	□□□⊠	□□□⊠	□⊠□□	□□□□	□□□⊠
inline-functions	□⊠□□	⊠□⊠⊠	□□□⊠	□⊠□□	□□⊠□	□⊠□□	□□□□	□□□⊠	□□□□	□⊠⊠□
inline-small-functions	□□□□	⊠□⊠□	□□□□	⊠□⊠□	□□□□	□□□□	□□⊠□	□□□□	□⊠⊠⊠	⊠□□⊠
ipa-cp-clone	⊠⊠⊠⊠	⊠⊠⊠□	□□□□	□⊠⊠□	□□□□	□□□□	⊠⊠□□	⊠⊠□□	□□□□	□⊠□□
ipa-profile	⊠⊠⊠⊠	□□⊠□	□□⊠⊠	⊠⊠□□	□⊠□□	□□⊠□	⊠⊠⊠□	□⊠□□	□⊠□□	□⊠□⊠
ipa-pure-const	⊠⊠□⊠	⊠□⊠□	□□□□	□□□□	□□□⊠	□□□⊠	⊠□⊠□	□⊠□□	□□⊠⊠	⊠⊠⊠⊠
ipa-reference	□⊠□□	⊠□□⊠	□⊠□⊠	□⊠□□	□□□⊠	□⊠⊠□	□□□□	□□□□	□⊠⊠□	□⊠□□
ipa-sra	□⊠□⊠	⊠□⊠⊠	□□□□	□⊠□□	⊠□□⊠	□□□□	⊠⊠□□	□⊠□□	□□□□	□□□□
merge-constants	□□□⊠	□⊠□□	□□□⊠	□⊠□□	□□□⊠	⊠⊠⊠⊠	⊠⊠□□	□□□⊠	□□⊠⊠	□□⊠□
omit-frame-pointer	□□□⊠	□□□⊠	⊠⊠⊠⊠	□□□□	□□□□	□□□⊠	□□□⊠	⊠⊠⊠□	□□□□	□□□□


Optimisation
2dfir
blowfish
crc32
cubic
dijkstra
fdct
matmult-float
matmult-int
rijndael
sha
optimize-sibling-calls
partial-inlining
peephole2
predictive-commoning
regmove
reorder-blocks
reorder-functions
rerun-cse-after-loop
sched-interblock
sched-spec
schedule-insns
schedule-insns2
split-wide-types
strict-aliasing
strict-overflow
thread-jumps
tree-bit-ccp
tree-builtin-call-dce
tree-ccp
tree-ch
tree-copyrename
tree-dce
tree-dominator-opts
tree-dse
tree-forwprop
tree-fre
tree-partial-pre
tree-phiprop
tree-pre
tree-pta
tree-slsr
tree-sra
tree-switch-conversion
tree-tail-merge
tree-ter
tree-vectorize
tree-vrp
unit-at-a-time
unswitch-loops
vect-cost-model

Glossary

AST Abstract Syntax Tree. A tree-like representation of the source code after parsing. 24, 50

basic block A straight-line sequence of instructions where the only possible entry is to the first instruction and the only exit is after the last instruction. 56

CFG The Control Flow Graph (CFG) is formed of all the basic blocks and their edges. 71

constant pool An section of memory embedded within code that stores locally accessed constants/data. 57

dominator A basic block, A, dominates another block, B, if all paths of execution must go through block A to reach block B. 36, 109

dominator tree A tree of nodes where a node's parent is the immediate dominator of the node. 36

DVFS Dynamic Voltage and Frequency Scaling. 14, 28, 91, 94

fractional factorial design A statistical technique to reduce the number of tests needed when it is expected that there are few higher-order interactions. 32, 33, 36, 81, 90

higher-order interactions Interactions which occur which a combination of two or more parameters (optimisations) are enabled. 32

hill climbing An objective-maximisation method where solutions immediately adjacent to the current are explored and the best one selected. This is repeated iteratively to find the maximum. 26, 27

ILP Integer Linear Programming is a maximisation or minimisation problem expressed in terms of constraints where variables are integer. 48, 49, 72, 75

IR Intermediate Representation. A simplified representation of the sourcecode suitable for optimisation. 24, 50

NRMSD Normalised Root Mean Square Deviation. Compare two sequences of values. Defined as $\frac{\sqrt{\frac{\sum_i^n (x_i - y_i)^2}{n}}}{x_{max} - x_{min}}$ where x_i and y_i are the sequences of values. 59

SIMD Single Instruction Multiple Data. 2, 28

This page is intentionally blank.

Bibliography

[1] T. Patyk, H. Hannula, P. Kellomaki, and J. Takala. "Energy consumption reduction by automatic selection of compiler options". In: *2009 International Symposium on Signals, Circuits and Systems*. IEEE, July 2009, pp. 1–4. ISBN: 978-1-4244-3785-6. DOI: 10.1109/ISSCS.2009.5206106.

[2] A. Parikh, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. "Instruction scheduling based on energy and performance constraints". In: *Proceedings. IEEE Computer Society Workshop on VLSI*. IEEE Comput. Soc, 2000, pp. 37–42. ISBN: 0-7695-0534-1. DOI: 10.1109/IWV.2000.844527.

[3] S. Wu and S. Li. "Instruction Selection for ARM/Thumb Processors Based on a Multi-objective Ant Algorithm". In: *Computer Science - Theory and Applications*. Lecture Notes in Computer Science 3967.90207019 (2006). Ed. by D. Grigoriev, J. Harrison, and E. A. Hirsch, pp. 641–651. DOI: 10.1007/11753728.

[4] A. Seth, R. B. Keskar, and R. Venugopal. "Algorithms for energy optimization using processor instructions". In: *CASES '01 Proceedings of the 2001 international conference on Compilers, Architecture, and Synthesis for Embedded Systems*. New York, New York, USA: ACM, 2001, p. 195. ISBN: 1581133995. DOI: 10.1145/502251.502252.

[5] M. Verma, L. Wehmeyer, and P. Marwedel. "Dynamic Overlay of Scratchpad Memory for Energy Minimization". In: *International Conference on Hardware/software Codesign and System Synthesis*. ACM, 2004. ISBN: 1581139373. DOI: 10.1109/CODESS.2004.240826.

[6] R. Wilhelm, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, and R. Heckmann. "The worst-case execution-time problem - overview of methods and survey of tools". In: *ACM Transactions on Embedded Computing Systems* 7.3 (Apr. 2008), pp. 1–53. ISSN: 15399087. DOI: 10.1145/1347375.1347389.

[7] M. Valluri and L. K. John. "Is compiling for performance == compiling for power?" In: *Proceedings of the 5th Annual Workshop on Interaction between Compilers and Computer Architectures*. 2001. DOI: 10.1007/978-1-4757-3337-2_6.

[8] M. E. A. Ibrahim, M. Rupp, and S. E.-D. Habib. "Compiler-based optimizations impact on embedded software power consumption". In: *2009 Joint IEEE North-East Workshop on Circuits and Systems and TAISA Conference*. Ieee, June 2009, pp. 1–4. ISBN: 978-1-4244-4573-8. DOI: 10.1109/NEWCAS.2009.5290480.

[9] S. V. Gheorghita, H. Corporaal, and T. Basten. "Using iterative compilation to reduce energy consumption". In: *Proceedings of the 10th Annual Conference of the Advanced School for Computing and Imaging* (2004). URL: http://www.ics.ele.tue.nl/~epicurus/publications/asci04svg.pdf.

[10] M. E. A. Ibrahim, M. Rupp, and H. A. H. Fahmy. "Code transformations and SIMD impact on embedded software energy/power consumption". In: *2009 International Conference on Computer Engineering & Systems*. IEEE, Dec. 2009, pp. 27–32. ISBN: 978-1-4244-5842-4. DOI: 10.1109/ICCES.2009.5383317.

[11] V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee. "Instruction level power analysis and optimization of software". In: *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 13.2-3 (1996), pp. 223–238. ISSN: 0922-5773. DOI: 10.1007/BF01130407.

[12] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. "An accurate and fine grain instruction-level energy model supporting software optimizations". In: *Proceedings of PATMOS*. 2001. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.6971%5C&rep=rep1%5C&type=pdf.

[13] S. Kerrison and K. Eder. "Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor". In: *ACM Transactions on Embedded Computing Systems* 14.3 (Apr. 2015), pp. 1–25. ISSN: 15399087. DOI: 10.1145/2700104.

[14] Y.-H. Park, S. Pasricha, F. J. Kurdahi, and N. D. Dutt. "A Multi-Granularity Power Modeling Methodology for Embedded Processors". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19.4 (Apr. 2011), pp. 668–681. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2009.2039153.

[15] U. Liqat, S. Kerrison, S. Alejandro, K. Giorgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, and K. Eder. "Energy Consumption Analysis of Programs Based on XMOS ISA-Level Models". In: *23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*. Ed. by G. Gupta and R. Peña. Vol. 8901. Lecture Notes in Computer Science. Cham: Springer International Publishing, Sept. 2014. ISBN: 978-3-319-14124-4. DOI: 10.1007/978-3-319-14125-1.

[16] R. Jayaseelan and T. Mitra. "Estimating the Worst-Case Energy Consumption of Embedded Software". In: *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. IEEE, 2006, pp. 81–90. ISBN: 0-7695-2516-4. DOI: 10.1109/RTAS.2006.17.

[17] C. Belleudy. "Optimization of Energy Consumption". In: *Real-Time Systems Scheduling 1*. Ed. by M. Chetto. Hoboken, NJ, USA: John Wiley & Sons, Inc., Aug. 2014. Chap. 6, pp. 231–267. ISBN: 9781118984413. DOI: 10.1002/9781118984413.

[18] F. Yao, A. Demers, and S. Shenker. "A scheduling model for reduced CPU energy". In: *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE Comput. Soc. Press, 1995, pp. 374–382. ISBN: 0-8186-7183-1. DOI: 10.1109/SFCS.1995.492493.

[19] Y. S. Shao and D. Brooks. "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor". In: *International Symposium on Low Power Electronics and Design (ISLPED)*. Ieee, Sept. 2013, pp. 389–394. ISBN: 978-1-4799-1235-3. DOI: 10.1109/ISLPED.2013.6629328.

[20] M. Younis, M. Youssef, and K. Arisha. "Energy-aware routing in cluster-based sensor networks". In: *Proceedings. 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*. 410. IEEE Comput. Soc, 2002, pp. 129–136. ISBN: 0-7695-1840-0. DOI: 10.1109/MASCOT.2002.1167069.

[21] Y. Lee and S. Kim. "DRAM energy reduction by prefetching-based memory traffic clustering". In: *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI - GLSVLSI '11* (2011), p. 103. DOI: 10.1145/1973009.1973031.

[22] L. Chandra and S. Roy. "Estimation of energy consumed by software in processor caches". In: *2008 IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, Apr. 2008, pp. 21–24. ISBN: 978-1-4244-1616-5. DOI: 10.1109/VDAT.2008.4542403.

[23] J. Nunez-Yanez and G. Lore. "Enabling accurate modeling of power and energy consumption in an ARM-based System-on-Chip". In: *Microprocessors and Microsystems* 37.3 (May 2013), pp. 319–332. ISSN: 01419331. DOI: 10.1016/j.micpro.2012.12.004.

[24] L. N. Chakrapani, P. Korkmaz, V. J. Mooney III, K. V. Palem, K. Puttaswamy, and W. F. Wong. "The emerging power crisis in embedded processors: what can a (poor) compiler do?" In: *Proceedings of the 2001 international conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, 2001. ISBN: 1581133995. DOI: 10.1145/502217.502246.

[25] V. Tiwari, S. Malik, and A. Wolfe. "Compilation techniques for low energy: an overview". In: *Proceedings of 1994 IEEE Symposium on Low Power Electronics*. IEEE, 1994, pp. 38–39. ISBN: 0-7803-1953-2. DOI: 10.1109/LPE.1994.573195.

[26] S. Woo, J. Yoon, and J. Kim. "Low-power instruction encoding techniques". In: *SOC Design Conference* (2001). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.17.9233%5C&rep=rep1%5C&type=pdf.

[27] S. Manne, A. Klauser, and D. Grunwald. "Pipeline gating: speculation control for energy reduction". In: *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)* (1998), pp. 132–141. DOI: 10.1109/ISCA.1998.694769.

[28] W. Zhang, J. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. "Compiler-directed instruction cache leakage optimization". In: *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.* (2002), pp. 208–218. DOI: 10.1109/MICRO.2002.1176251.

[29] X. Guan and Y. Fei. "Register file partitioning and recompilation for register file power reduction". In: *ACM Transactions on Design Automation of Electronic Systems* 15.3 (May 2010), pp. 1–30. ISSN: 10844309. DOI: 10.1145/1754405.1754409.

[30] D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: a framework for architectural-level power analysis and optimizations". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture* (2000). DOI: 10.1145/342001.339657.

[31] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. *CACTI 6.0: A tool to understand large caches*. Tech. rep. University of Utah, 2009. URL: http://www.cs.utah.edu/~rajeev/cacti6/cacti6-tr.pdf.

[32] J. Pallister, S. Hollis, and J. Bennett. "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms". 2013. URL: http://arxiv.org/abs/1308.5174.

[33] J. Pallister, S. J. Hollis, and J. Bennett. "Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms". In: *The Computer Journal* 58.1 (Nov. 2013), pp. 95–109. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxt129.

[34] J. Pallister, K. Eder, S. J. Hollis, and J. Bennett. "A high-level model of embedded flash energy consumption". In: *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems - CASES '14*. New York, New York, USA: ACM Press, 2014, pp. 1–9. ISBN: 9781450330503. DOI: 10.1145/2656106.2656108.

[35] J. Pallister, K. Eder, and S. J. Hollis. "Optimizing the flash-RAM energy trade-off in deeply embedded systems". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* abs/1406.0 (Feb. 2015), pp. 115–124. DOI: 10.1109/CGO.2015.7054192. arXiv: 1406.0403.

[36] Xilinx. *Xilinx Software Development Kit (SDK) User Guide*. Tech. rep. Xilinx, 2014.

[37] J. Constantin, A. Bonetti, A. Teman, L. Duch, P. Garcia, and D. Atienza. *SCoRPiO Project - D4.4: Mechanisms for runtime fault detection and software controlled hardware reconfiguration*. Tech. rep. EPFL, 2015.

[38] J. Constantin, L. Wang, G. Karakonstantis, A. Chattopadhyay, and A. Burg. "Exploiting Dynamic Timing Margins in Microprocessors for Frequency-Over-Scaling with Instruction-Based Clock Adjustment". In: *Proceedings of the 2015 Design, Automation & Test in Europe*. 2015, pp. 381–386. ISBN: 9783981537048.

[39] P. Wägemann, T. Distler, T. Hönig, V. Sieh, and W. Schröder-preikschat. "GenE : A benchmark generator for WCET a nalysis". In: *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*. 2015, pp. 33–43. DOI: 10.4230/OASIcs.WCET.2015.33.

[40] J. Bennett, S. J. Hollis, K. Eder, O. Ray, J. Pallister, S. Cook, E. Jones, and C. Blackmore. *MAGEEC*. 2015. URL: http://mageec.org/.

[41] Free Software Foundation. *The GNU Compiler Collection*. 2014. URL: http://gcc.gnu.org/.

[42] *The LLVM Compiler Infrastructure*. URL: http://llvm.org/.

[43] J. Pallister, S. Kerrison, J. Morse, and K. Eder. "Data dependent energy modelling: A worst case perspective". In: *CoRR* abs/1505.0 (2015). URL: http://arxiv.org/abs/1505.03374.

[44] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder. "Static analysis of energy consumption for LLVM IR programs". In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '15. New York, NY, USA: ACM, 2015, pp. 12–21. ISBN: 978-1-4503-3593-5. DOI: 10.1145/2764967.2764974.

[45] C.-H. Hsu and U. Kremer. "Compiler-Directed Dynamic CPU Frequency and Voltage Scaling". In: *Designing Embedded Processors*. Ed. by J. Henkel and S. Parameswaran. Springer Netherlands, 2007, pp. 305–323. DOI: 10.1007/978-1-4020-5869-1_14.

[46] M. R. Guthaus and J. S. Ringenberg. "MiBench: A free, commercially representative embedded benchmark suite". In: *IEEE International Workshop on Workload Characterization (WWC-4)*. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.15.

[47] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. "MediaBench II video: Expediting the next generation of video systems research". In: *Microprocessors and Microsystems* 33.4 (June 2009), pp. 301–318. ISSN: 01419331. DOI: 10.1016/j.micpro.2009.02.010.

[48] J. J. Dongarra, P. Luszczek, and A. Petitet. "The LINPACK Benchmark: past, present and future". In: *Concurrency and Computation: Practice and Experience* 15.9 (Aug. 2003), pp. 803–820. ISSN: 1532-0626. DOI: 10.1002/cpe.728.

[49] R. P. Weicker. "Dhrystone benchmark: rationale for version 2 and measurement rules". In: *ACM SIGPLAN Notices* 23.8 (1988). DOI: 10.1145/47907.47911.

[50] EEMBC. *ULPBench*. 2014. URL: http://www.eembc.org/ulpbench/.

[51] C. Bienia, S. Kumar, and K. Li. "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors". In: *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on* (Oct. 2008), pp. 47–56. DOI: 10.1109/IISWC.2008.4636090.

[52] G. Fursin et al. "Milepost GCC: machine learning enabled self-tuning compiler". In: *International Journal of Parallel Programming* (2011), pp. 1–31. URL: http://www.springerlink.com/index/D753R27550257252.pdf.

[53] S. M. Z. Iqbal, Y. Liang, and H. Grahn. "ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems". In: *IEEE Computer Architecture Letters* 9.2 (Feb. 2010), pp. 45–48. ISSN: 1556-6056. DOI: 10.1109/L-CA.2010.14.

[54] M. Weiland and N. Johnson. "Benchmarking for power consumption monitoring". In: *Computer Science - Research and Development* 30.2 (July 2014), pp. 155–163. ISSN: 1865-2034. DOI: 10.1007/s00450-014-0260-1.

[55] V. Zivojnovic, J. M. Velarde, C. Schlager, and H. Meyr. "DSPstone: A DSP-oriented benchmarking methodology". In: *Proc. of ICSPAT*. 1994, pp. 715–720.

[56] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. "The Mälardalen WCET benchmarks, past, present and future". In: *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*. 2010. DOI: 10.4230/OASIcs.WCET.2010.136.

[57] H. Blume, D. Becker, L. Rotenberg, M. Botteck, J. Brakensiek, and T. Noll. "Hybrid functional- and instruction-level power modeling for embedded and heterogeneous processor architectures". In: *Journal of Systems Architecture* 53.10 (Oct. 2007), pp. 689–702. ISSN: 13837621. DOI: 10.1016/j.sysarc.2007.01.002.

[58] S. Lee, A. Ermedahl, S. L. Min, and N. Chang. "An accurate instruction-level energy consumption model for embedded risc processors". In: *ACM SIGPLAN Notices* (2001). URL: http://dl.acm.org/citation.cfm?id=384201.

[59] H.-h. S. Lee, J. B. Fryman, A. U. Diril, and Y. S. Dhillon. "The elusive metric for low-power architecture research". In: *Proceedings of the Workshop on Complexity-Effective Design*. 2003. URL: http://arch.ece.gatech.edu/pub/wced03.pdf.

[60] R. Gonzalez and M. Horowitz. "Energy dissipation in general purpose processors". In: *1995 IEEE Symposium on Low Power Electronics. Digest of Technical Papers*. IEEE, 1995, pp. 12–13. ISBN: 0-7803-3036-6. DOI: 10.1109/LPE.1995.482411.

[61] J. H. Laros III, K. Pedretti, S. M. Kelly, W. Shu, K. Ferreira, J. Vandyke, and C. Vaughan. "Energy Delay Product". In: *Energy-Efficient High Performance Computing*. SpringerBriefs in Computer Science, 2013, pp. 51–55. ISBN: 978-1-4471-4491-5. DOI: 10.1007/978-1-4471-4492-2_8.

[62] Z. Chen, X. Liu, R. Zhang, and H. Liu. "An Automotive Electronic Throttle Testing Equipment Based on STM32". In: *2014 International Symposium on Computer, Consumer and Control* (June 2014), pp. 478–481. DOI: 10.1109/IS3C.2014.131.

[63] A. E. Kalman. *Hardware and software design of an MSP430-based satellite using an RTOS*. Tech. rep. Texas Instruments, 2004, pp. 1–46.

[64] A. Huang and S. Cross. *Novena Battery Board*. 2014. URL: http://www.kosagi.com/w/index.php?title=Novena%5C_Main%5C_Page%5C#Battery%5C_board.

[65] P. Lepek. *Designing next-generation key fobs*. Tech. rep. Atmel, 2010, pp. 15–20. URL: http://www.atmel.com/Images/Article%5C_AC7%5C_Designing-Next-Generation-Key-Fobs.pdf.

[66] B. Smith. "ARM and Intel Battle over the Mobile Chip's Future". In: *Computer* 41.5 (May 2008), pp. 15–18. ISSN: 0018-9162. DOI: 10.1109/MC.2008.142.

[67] R. Rebe. *Openbench*. 2012. URL: http://www.exactcode.com/site/open%5C_source/openbench/.

[68] J. L. Henning. "SPEC CPU2006 benchmark descriptions". In: *ACM SIGARCH Computer Architecture News* 34.4 (Sept. 2006), pp. 1–17. ISSN: 01635964. DOI: 10.1145/1186736.1186737.

[69] ARM Limited. "Cortex-M0 Technical Reference Manual". In: (2009).

[70] J. Yiu. *The Definitive Guide to the ARM Cortex-M3*. 2nd. Newnes, 2010. ISBN: 978-1-85617-963-8.

[71] Atmel. *Atmel 8-bit Microcontroller*. 2013.

[72] Atmel. *Atmel AVR1923: XMEGA-A3BU Xplained Hardware User Guide*. Tech. rep. 2012, pp. 1–19.

[73] Microchip. *PIC32MX Family*. Tech. rep. 2012, pp. 1–64.

[74] Texas Instruments. *MSP430F5529 Mixed signal Microcontroller*. Tech. rep. March 2009. 2013.

[75] Texas Instruments. *MSP430FR5739 Mixed-Signal Microcontroller*. Tech. rep. 2014.

[76] Y. Kato, H. Tanaka, K. Isogai, K. Kaibara, Y. Kaneko, Y. Shimada, M. Brubaker, J. Celinska, L. D. McMillan, and C. A. P. de Araujo. "Embedded FeRAM Challenges in the 65-nm Technology Node and Beyond". In: *2006 IEEE International Symposium on the Applications of Ferroelectrics*. IEEE, July 2006, pp. 81–84. ISBN: 978-1-4244-1331-7. DOI: 10.1109/ISAF.2006.4387838.

[77] G. Coley. *BeagleBone Rev A3 System Reference Manual*. 2011. URL: http://download.tigal.com/beagle/BeagleBone%5C_SRM%5C_A6%5C_0%5C_1.pdf.

[78] Texas Instruments. *AM335x ARM Cortex-A8 Microprocessors*. 2012.

[79] Adapteva. *E16G301 Epiphany 16-core microprocessor datasheet*. 2013. URL: http://www.adapteva.com/wp-content/uploads/2013/06/e16g301%5C_datasheet%5C_3.13.6.14.pdf.

[80] D. May. *The XMOS XS1 Architecture*. ISBN: 9781907361012.

[81] P. A. Kulkarni, D. B. Whalley, and G. S. Tyson. "Evaluating Heuristic Optimization Phase Order Search Algorithms". In: *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, Mar. 2007, pp. 157–169. ISBN: 0-7695-2764-7. DOI: 10.1109/CGO.2007.9.

[82] R. Eigenmann. "Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning". In: *International Symposium on Code Generation and Optimization (CGO'06)*. ii. IEEE, 2006, pp. 319–332. ISBN: 0-7695-2499-0. DOI: 10.1109/CGO.2006.38.

[83] E. Schkufza, R. Sharma, and A. Aiken. "Stochastic superoptimization". In: *Architectural Support for Programming Languages and Operating Systems*. New York, New York, USA: ACM Press, 2013, p. 305. ISBN: 9781450318709. DOI: 10.1145/2451116.2451150.

[84] K. Chow and Y. Wu. "Feedback-directed selection and characterization of compiler optimizations". In: *Proceedings of the Second Workshop on Feedback-Directed Optimization*. 1999, pp. 1–10. URL: http://cseweb.ucsd.edu/users/calder/fdo/fdo2/papers/fdo2-wu.ps.

[85] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. John Wiley & Sons, 1978, pp. 374–418. ISBN: 0-471-09315-7.

[86] S.-c. Lin, C.-k. Chang, and N.-w. Lin. "Automatic selection of GCC optimization options using a gene weighted genetic algorithm". In: *Computer Systems Architecture Conference, 2008. ACSAC 2008. 13th Asia-Pacific* (Aug. 2008), pp. 1–8. DOI: 10.1109/APCSAC.2008.4625477.

[87] K. D. Cooper, P. J. Schielke, and D. Subramanian. "Optimizing for reduced code space using genetic algorithms". In: *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems - LCTES '99*. New York, New York, USA: ACM Press, 1999, pp. 1–9. ISBN: 1581131364. DOI: 10.1145/314403.314414.

[88] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. "Compiler techniques for code compaction". In: *ACM Transactions on Programming Languages and Systems* 22.2 (Mar. 2000), pp. 378–415. ISSN: 01640925. DOI: 10.1145/349214.349233.

[89] A. Nisbet. "GAPS : Genetic Algorithm Optimised Parallelisation". In: *Proc. Workshop on Profile and Feedback Directed Compilation*. 1998. DOI: 10.1007/BFb0037253.

[90] N. Azeemi. "Multicriteria Energy Efficient Source Code Compilation for Dependable Embedded Applications". In: *2006 Innovations in Information Technology*. IEEE, Nov. 2006, pp. 1–5. ISBN: 1-4244-0673-0. DOI: 10.1109/INNOVATIONS.2006.301963.

[91] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. "Post-compiler software optimization for reducing energy". In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems - ASPLOS '14* (2014), pp. 639–652. DOI: 10.1145/2541940.2541980.

[92] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. "Rapidly Selecting Good Compiler Optimizations using Performance Counters". In: *International Symposium on Code Generation and Optimization (CGO'07)*. Ieee, Mar. 2007, pp. 185–197. ISBN: 0-7695-2764-7. DOI: 10.1109/CGO.2007.32.

[93] S. Kulkarni and J. Cavazos. "Mitigating the compiler optimization phase-ordering problem using machine learning". In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications* (2012), pp. 1–16. URL: http://dl.acm.org/citation.cfm?id=2384616.2384628.

[94] K. O. Stanley and R. Miikkulainen. "Efficient Reinforcement Learning through Evolving Neural Network Topologies". In: *Genetic and Evolutionary Computation Conference*. New York, NY, USA, 2002, pp. 569–577.

[95] G. Magklis, M. Scott, and D. Albonesi. "The energy impact of aggressive loop fusion". In: *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* IEEE, 2004, pp. 153–164. ISBN: 0-7695-2229-7. DOI: 10.1109/PACT.2004.1342550.

[96] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. "Reducing energy consumption by dynamic copying of instructions onto onchip memory". In: *Proceedings of the 15th international symposium on System Synthesis - ISSS '02*. New York, New York, USA: ACM Press, 2002, p. 213. ISBN: 1581135769. DOI: 10.1145/581199.581247.

[97] Y. Ishitobi, T. Ishihara, and H. Yasuura. "Code and Data Placement for Embedded Processors with Scratchpad and Cache Memories". In: *Journal of Signal Processing Systems* 60.2 (Nov. 2008), pp. 211–224. ISSN: 1939-8018. DOI: 10.1007/s11265-008-0306-3.

[98] D. A. Ortiz and N. G. Santiago. "Impact of source code optimizations on power consumption of embedded systems". In: *2008 Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference* (June 2008), pp. 133–136. DOI: 10.1109/NEWCAS.2008.4606339.

[99] C. F. J. Wu and M. Hamada. *Experiments: Planning, analysis, and parameter design optimization*. New York: Wiley, 2000, p. 112. ISBN: 978-0471255116.

[100] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. "Practical exhaustive optimization phase order exploration and evaluation". In: *ACM Transactions on Architecture and Code Optimization* 6.1 (Mar. 2009), pp. 1–36. ISSN: 15443566. DOI: 10.1145/1509864.1509865.

[101] ARM Limited. *Cortex-M3 Technical Reference Manual*. 2006.

[102] O. Zendra. "Memory and compiler optimizations for low-power and -energy". In: (Oct. 2006). arXiv: 0610028 [cs]. URL: http://arxiv.org/abs/cs/0610028.

[103] C.-h. Hsu and U. Kremer. "The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction". In: *ACM SIGPLAN Notices* 38.5 (May 2003), p. 38. ISSN: 03621340. DOI: 10.1145/780822.781137.

[104] D. Zhurikhin, A. Belevantsev, and A. Avetisyan. "Evaluating power-aware optimizations within GCC compiler". In: *GCC Research Opportunities Workshop (GROW '09)* (2009). URL: http://www.doc.ic.ac.uk/~phjk/GROW09/papers/06-PowerBelevantsev.pdf.

[105] K. Zhang, T. Zhang, and S. Pande. "Binary translation to improve energy efficiency through post-pass register re-allocation". In: *Proceedings of the fourth ACM international conference on Embedded software - EMSOFT '04*. New York, New York, USA: ACM Press, 2004, p. 74. ISBN: 1581138601. DOI: 10.1145/1017753.1017769.

[106] A. G. M. Cilio and H. Corporaal. "Global Variable Promotion: Using Registers to Reduce Cache Power Dissipation". In: *Compiler Construction*. Ed. by R. N. Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, Mar. 2002, pp. 247–261. ISBN: 978-3-540-43369-9. URL: http://link.springer.com/10.1007/3-540-45937-5.

[107] P. Petrov and A. Orailoglu. "Compiler-based register name adjustment for low-power embedded processors". In: *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No.03CH37486)*. IEEE, 2003, pp. 523–527. ISBN: 1-58113-762-1. DOI: 10.1109/ICCAD.2003.159734.

[108] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. "Techniques for low energy software". In: *Proceedings of the 1997 international symposium on Low power electronics and design - ISLPED '97*. New York, New York, USA: ACM Press, 1997, pp. 72–75. ISBN: 0897919033. DOI: 10.1145/263272.263286.

[109] H. Tomiyama and T. Ishihara. "Instruction scheduling for power reduction in processor-based system design". In: *Proceedings of the conference on Design, automation and test in Europe*. IEEE, 1998, pp. 855–860. URL: http://dl.acm.org/citation.cfm?id=368439.

[110] M. C. Toburen, T. M. Conte, and M. Reilly. "Instruction scheduling for low power dissipation in high performance microprocessors". In: *Proceedings of the 1998 Power Driven Micro-architecture Workshop*. 1998. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.62.2534%5C&rep=rep1%5C&type=pdf.

[111] A. W. Min, R. Wang, J. Tsai, M. A. Ergin, and T.-Y. C. Tai. "Improving energy efficiency for mobile platforms by exploiting low-power sleep states". In: *Proceedings of the 9th conference on Computing Frontiers - CF '12*. New York, New York, USA: ACM Press, 2012, p. 133. ISBN: 9781450312158. DOI: 10.1145/2212908.2212928.

[112] V. Venkatachalam and M. Franz. "Power reduction techniques for microprocessor systems". In: *ACM Computing Surveys* 37.3 (Sept. 2005), pp. 195–237. ISSN: 03600300. DOI: 10.1145/1108956.1108957.

[113] L. Wehmeyer and P. Marwedel. "Scratchpad Memory Optimizations". In: *Fast , Efficient and Predictable Memory Accesses: Optimization Algorithms for Memory Architecture Compilation*. 1st ed. Dordrecht, The Netherlands: Springer Netherlands, 2006. Chap. 4, pp. 89–169. ISBN: 1402048211.

[114] S. Steinke, L. Wehmeyer, and P. Marwedel. "Assigning program and data objects to scratchpad for energy reduction". In: *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*. IEEE Comput. Soc, 2002, pp. 409–415. ISBN: 0-7695-1471-5. DOI: 10.1109/DATE.2002.998306.

[115] M. Kandemir, I. Kadayif, and U. Sezer. "Exploiting scratch-pad memory using Presburger formulas". In: *Proceedings of the 14th international symposium on Systems synthesis - ISSS '01* (2001), p. 7. DOI: 10.1145/500002.500004.

[116] L. Gauthier, T. Ishihara, H. Takase, H. Tomiyama, and H. Takada. "Minimizing inter-task interferences in scratch-pad memory usage for reducing the energy consumption of multi-task systems". In: *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems - CASES '10*. New York, New York, USA: ACM Press, 2010, p. 157. ISBN: 9781605589039. DOI: 10.1145/1878921.1878945.

[117] M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu. "Compiler-directed scratch pad memory optimization for embedded multiprocessors". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.3 (Mar. 2004), pp. 281–287. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2004.824299.

[118] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. "EnerJ: Approximate Data Types for Safe and General Low-Power Computation". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11*. New York, New York, USA: ACM Press, 2011, p. 164. ISBN: 9781450306638. DOI: 10.1145/1993498.1993518.

[119] A. a. Eltawil, M. Engel, B. Geuskens, A. K. Djahromi, F. J. Kurdahi, P. Marwedel, S. Niar, and M. a.R. Saghir. "A survey of cross-layer power-reliability tradeoffs in multi and many core systems-on-chip". In: *Microprocessors and Microsystems* 37.8 (Nov. 2013), pp. 760–771. ISSN: 01419331. DOI: 10.1016/j.micpro.2013.07.008.

[120] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. "Characterizing flash memory: Anomalies, Observations, and Applications". In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42*. New York, New York, USA: ACM Press, 2009, p. 24. ISBN: 9781605587981. DOI: 10.1145/1669112.1669118.

[121] V. Mohan, T. Bunker, L. Grupp, S. Gurumurthi, M. R. Stan, and S. Swanson. "Modeling Power Consumption of NAND Flash Memories Using FlashPower". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.7 (July 2013), pp. 1031–1044. ISSN: 0278-0070. DOI: 10.1109/TCAD.2013.2249557.

[122] S. Kim, K. Kwon, C. Kim, C. Jang, J. Lee, and S. L. Min. "Demand Paging Techniques for Flash Memory Using Compiler Post-Pass Optimizations". In: *ACM Transactions on Embedded Computing Systems* 10.4 (Nov. 2011), pp. 1–29. ISSN: 15399087. DOI: 10.1145/2043662.2043664.

[123] H.-w. Park, S. Park, and M.-m. Sim. "Dynamic Code Overlay of SDF-Modeled Programs on Low-end Embedded Systems". In: *Proceedings of the Design Automation & Test in Europe Conference* (2006), pp. 1–2. DOI: 10.1109/DATE.2006.243836.

[124] ARM Limited. *What are Overlays and how are they used?* 2011. URL: http://infocenter.arm.com/help/topic/com.arm.doc.faqs/ka4234.html.

[125] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. "Drowsy caches: simple techniques for reducing leakage power". In: *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE Comput. Soc, 2002, pp. 148–157. ISBN: 0-7695-1605-X. DOI: 10.1109/ISCA.2002.1003572.

[126] B. Calder, C. Krintz, S. John, and T. Austin. "Cache-conscious data placement". In: *ACM SIGPLAN Notices* 33.11 (Nov. 1998), pp. 139–149. ISSN: 03621340. DOI: 10.1145/291006.291036.

[127] Zeptobars. *STM32F103VGT6 : Weekend die-shot*. 2012. URL: http://zeptobars.ru/en/read/STM-STM32F103VGT6.

[128] P. Cavaleri, B. Leconte, S. Zink, and J. Devin. *Page-erasable flash memory*. 2004.

[129] STMicroelectronics. *High density NAND flash memories*. 2005.

[130] D. Brylow, N. Damgaard, and J. Palsberg. "Static checking of interrupt-driven software". In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. IEEE Comput. Soc, 2001, pp. 47–56. ISBN: 0-7695-1050-7. DOI: 10.1109/ICSE.2001.919080.

[131] Free Software Foundation. *GNU Linear Programming Kit, Version 4.52*. 2014. URL: http://www.gnu.org/software/glpk/glpk.html.

[132] Y. Joo, Y. Cho, D. Shin, J. Park, and N. Chang. "An energy characterization platform for memory devices and energy-aware data compression for multilevel-cell flash memory". In: *ACM Transactions on Design Automation of Electronic Systems* 13.3 (July 2008), pp. 1–29. ISSN: 10844309. DOI: 10.1145/1367045.1367052.

[133] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen. "Tuning compiler optimizations for simultaneous multithreading". In: *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE Comput. Soc, 1997, pp. 114–124. ISBN: 0-8186-7977-8. DOI: 10.1109/MICRO.1997.645803.

[134] ARM Limited. *big.LITTLE Technology : The Future of Mobile Same architecture but different micro-architectures*. Tech. rep. ARM Limited, 2013, pp. 1–12. URL: http://www.arm.com/files/pdf/big%5C_LITTLE%5C_Technology%5C_the%5C_Futue%5C_of%5C_Mobile.pdf.

[135] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. 2nd ed. Addison-Wesley, 2007, pp. 583–706.

[136] S. S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

[137] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. "Conversion of control dependence to data dependence". In: *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '83*. New York, New York, USA: ACM Press, 1983, pp. 177–189. ISBN: 0897910907. DOI: 10.1145/567067.567085.

[138] P. Bonzini and L. Pozzi. "Code transformation strategies for extensible embedded processors". In: *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems - CASES '06* (2006), p. 242. DOI: 10.1145/1176760.1176791.

[139] Free Software Foundation. *GCC: Options that control optimization*. 2014. URL: https://gcc.gnu.org/onlinedocs/gcc-4.9.1/gcc/Optimize-Options.html.

[140] A. Tal. "Second-Order Predictive Commoning". In: *3rd Workshop on Compiler-Driven Performance*. Markham, ON, USA, 2004. URL: https://webdocs.cs.ualberta.ca/~amaral/cascon/CDP04/slides/tal.pdf.

This page is intentionally blank.

Index