Software approaches and optimisations for energy efficiency in embedded systems

James Pallister

Supervisor: Kerstin Eder
Second supervisor: Simon Hollis

November 29, 2013

Abstract

It is increasingly important to consider the energy consumption of software with the hardware it is running on. A large amount of software already exists for embedded platforms and the easiest method a developer can employ to increase their code's performance is by upgrading their compiler. To retain the same advantage for energy consumption, it is necessary to develop techniques and optimisations that the compiler can use to improve the code's energy efficiency in an automated way.

This report considers a variety of techniques that the compiler can use to reduce energy consumption, such as optimisation selection and ordering, novel optimisations for energy, superoptimisation, and specific hardware trade-offs (including RAM vs. flash). It is observed that each individual technique can reduce energy consumption and the argument is made that when all techniques are combined, a significant amount of energy could be saved by the compiler.

Contents

1

Introduction

Energy consumption is a primary design constraint in many embedded platforms, with battery life being one of the most important characteristics of the device. There are many hardware approaches to reducing the energy consumption at the silicon level, however, it is the software running on the hardware that ultimately controls the energy consumption.

There are two sources of energy consumption in a processor: static and dynamic power dissipation. The dynamic power is dependent on how often the transistors change state — usually a result of software. For example, a memory load instruction will cause the a value to be put on the address line, and associated control logic to toggle. The static power dissipation is leakage through each transistor: it occurs whether or not the transistor flips. This is more difficult to control from software — the only way to reduce it is either to run the software in less time and switch the transistor off, or change the operating voltage of the transistor.

The software defines which parts of the processor will be powered and which activities they will perform. This leads to the need for software approaches to reduce the energy consumption of the underlying hardware. The invasiveness of a software technique greatly affects its further re-use and take-up by industry. In this respect it is more reasonable to expect a developer to recompile their program with an energy efficient compiler than it is to expect them to completely re-architecture their existing application to follow an energy efficient paradigm. For this reason, compilers are well placed to have a wide impact on a large amount of software. E.g. an existing software library only requires recompilation to become $X\%$ more energy efficient. This report focuses on methods that the compiler can employ to reduce the energy consumption of the code being compiled.

Existing compiler optimisations can be effective at reducing energy consumption, however, the majority of this saving comes from the performance boost that these optimisations also bring. The performance boost reduces the running time, resulting in static power dissipation for a shorter amount of time. This also occurs frequently because of fewer instructions executed: less work to do results in lower energy consumption. The way that compiler optimisations combine is non-trivial with extreme differences in performance just from differently ordering optimisations. The selection and ordering of compiler optimisations will therefore have a large effect on the energy consumption. This report presents initial exploration into the selection and ordering of existing compiler optimisations.

While the majority of existing compiler optimisations improve energy consumption in proportion to execution time, it is possible to develop new optimisations that purely target energy consumption. Some of these exploit the fact that the source of dynamic energy consumption is the amount of bit-flips occurring in the pipeline, e.g. replacing $x = y \times 2$ with $x = y + y$ would reduce energy consumption if a multiply causes more bits to flip (as is

often the case). Some of these optimisations could be found by *superoptimisation* — a brute force search through possible instruction sequences until the optimal instruction sequence is found. Other optimisations for energy include exploiting resources in the hardware platform. If a hardware peripheral is available to do CRC calculations, this is likely much more energy efficient than the software equivalent. Several ideas for new compiler optimisations are presented in this report, along with the current progress on developing a superoptimiser for energy.

Once new optimisations have been developed, they must be combined in the correct order and with the correct other optimisations to achieve maximal efficacy. This is a non-trivial task, and it is currently unknown how several optimisations for energy would interact with each other.

In future, compilers will not be just optimising existing code. They will be able to take a holistic view of the entire platform, selecting the best optimisations and variants of libraries to minimise energy consumption. This could possibly utilise JIT compilation, where a code trace is noticed to be energy inefficient at runtime, and recompiled on the fly to produce a specific energy optimal version. The compiler, combined with runtime and static analysis will allow a system to dynamically optimise itself, simultaneously controlling the memory hierarchy, hardware peripherals, the processor's frequency and voltage to balance the trade-off between performance and energy consumption. The techniques detailed in this report are a step towards the compiler being able to optimise for energy and consider platform specific information to reduce energy consumption.

The next chapter of this report covers related work in this area. Then, follows contributions made to various areas, including ongoing work. Chapter 4 covers areas and ideas which could be developed in the future. Then, the final chapter presents the conclusion of this report, followed by my publications. The appendices list a plan for the remainder of my PhD, followed by a list of activities and events attended over the last year.

2

Related work

This chapter discusses work which has attempted to reduce energy consumption by using some form of software component. This sometimes takes the form of compiler optimisations targeting energy consumption, or transformations that can be made to the source code. This allows more efficient usage of the platform's resources and peripherals. Other techniques to reduce energy consumption have exploited hardware-software codesign, where specialised or efficient hardware components are designed along with the software required to exploit them.

This section first discusses architectural modifications that can be made to processor to enable energy-efficient execution. The following section discusses compiler optimisations — optimisations specifically for energy, as well as selection and ordering of existing compiler optimisations. The final section discusses how models of energy consumption can be constructed.

2.1 Architectural modifications

This section discusses modifications that can be made to a processor's architecture to reduce energy consumption. Many of these modifications also require software support to function correctly, however some are already present in many processors, such as scratch pad memory.

'Bus invert coding' is a technique which does not require software support [1]. This modification relies on the fact the it costs more energy to drive a high capacitance bus line than to have several more transistors to optimise the encoding. This technique adds an additional line to the bus, which indicates that the rest of the lines are inverted. This can minimise the average number of transitions on the bus by up to 25%.

Another method of reducing the amount of bit flips in a processor is to correctly choose the instruction encodings. Woo et al. [2] reduce the switching between op-codes for the MIPS architecture. The op-code switching is reduced 40–60%, however there is no analysis of the amount of reduction in energy this may lead to when taken in the context of a whole system.

Scratch pad memory has be explored extensively, as optimal placement of code and data items in this memory will reduce both execution time and energy consumption [3]. Scratch pad memories are also typically more efficient than cache's due the lack of extra logic needed by caches — the difficulty is that they must be controlled by software. In [4] code and data objects are moved by the compiler into scratch pad memory, and found to have a 43% saving in energy compared to an equivalent sized cache. This has also been explored in the case of a multi-task system [5].

Register file partitioning has been shown to reduce energy consumption [6]. Guan et al. noticed that for some embedded processors only 25% of the registers were used for the majority of the time. This led to a modification of the register file, where part of it could be disabled and put into a low power state. This was combined with an analysis to optimally choose which values should go into the hot and cold regions of the register file. Overall, roughly 50% of the register file power saving could be achieved with 5% slow down.

The majority of processors do not expose energy consumption characteristics to the programmer. Asanovic et al. [7] describe how additional instructions could be created which allow more energy-efficient operations to be executed, such as bypassing certain cache operations when data is known to be in cache. Energy expensive operations, such as shifting could also have more specific but energy efficient versions for frequent operands, trading silicon area for energy efficiency. Another development in this area exposes the bypass network between pipeline stages to the compiler, allowing sequences of instructions to be constructed that do not use the register file [8].

2.2 Compiler optimisations

This section discusses how compiler optimisations can be used to reduce the energy consumption of embedded systems.

2.2.1 Optimizing for energy

Many previous studies look at how to utilise *existing* optimisations to target energy consumption. However, all of these optimisations were written with the aim of reducing execution time, not energy consumption. Several other techniques have been proposed to develop optimisations that specifically target energy consumption.

An analysis of the techniques the compiler can perform to optimise for energy was carried out by Tiwari, Malik and Wolfe [9]. They identified several possible techniques that compilers could use to reduce the energy consumption of programs. They were:

- Reorder instructions to reduce switching.
- Reduce switching on address lines.
- Reduce memory accesses.
- Improve cache hits.
- Improve page hits.

The last three will also normally increase performance as well as reduce energy.

Several novel types of compiler optimisations have been proposed. Seth et al. [10] explored the possibility of using the compiler to insert `idle` instructions automatically, increasing the execution time up to a set limit.

Source code transformations and the use of the SIMD pipeline has been shown to have an effect on energy consumption and power dissipation in [11]. This study found that reordering array declarations could reduce the energy consumption, althoughan explanation of why this occurred was not given. Also found was that software loop pipelining had a huge effect on power dissipation, especially when the code was vectorised to use the SIMD pipelines.

Scheduling instructions to minimise the inter-instruction energy cost was evaluated to be a moderately effective method to reduce a program's energy consumption. Parikh et al. [12] examine six different instruction scheduling methods including, random, pure performance-based, pure energy-based and combinations of performance and energy. They find only

small differences in performance when these algorithms are applied. However, large savings in energy consumption are seen, provided the scheduling algorithm considers the energy cost of the instructions. There was not a large difference seen between the energy-considering algorithms, however random and pure performance consumed larger amounts of energy.

Exploiting differences in energy consumption between other function units has been suggested in [13], where it is noted that strength reduction may use a more efficient shifter rather than a power hungry multiplier. Other techniques have also been employed to reduce the energy cost of going to memory by accounting for the bit-width required by the variable being accessed [14].

Dynamic Voltage and Frequency Scaling (DVFS) can be used to change the amount of power and the time taken for operations to execute. This exploits the fact that $P \propto V^2$, where P is the power dissipation of the processor and V is the voltage the processor is currently using. However, by lowing the voltage, the transistors take longer to stabilise, and so the clock frequency must also be reduced. This leads to code being able to either execute slow and efficiently or fast and power-hungry. This effect has been exploited to reduce energy consumption while still meeting some timing constraints [15]. This allows larger energy savings — up to 70% — if timing constraints can be relaxed, or lower savings if an tighter schedule needs to be met.

2.2.2 Selection

Iterative compilation is used to explore the tile size and unroll factor parameters to the compiler in [16]. The effect on performance, energy and the energy delay product is shown to have a semi-regular pattern. 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.

Pan et al. [17] explore several approaches to optimisation selection (for performance), and propose a new method based on several state of the art techniques. Batch Elimination executions the application with optimisations individually disabled, then disabling all optimisations which decreased the performance. However, this approach does not always perform well, since it does not consider the interactions between optimisations. Another algorithm explored tries to iteratively find optimisations to eliminate, repeating the process until no optimisations with negative impacts are left.

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

Several studies explore the optimisation space in systematic way, allowing optimisation selection and general conclusions about the nature of the space to be made.

Chow et al. [18] take the approach of using fractional factorial design to choose a subset of tests to explore the effect of nine optimisations on an application's performance. Using this method, the number of runs was reduced to 32, instead of 512, allowing acceptable run times. Using the results, the set of optimisations was able to be modified, selecting a set which improved the performance over naively selecting every optimisation. A further conclusion of this study was that examining optimisation individually was a poor metric of how the optimisation would perform when combined with other optimisations.

Fractional factorial design [19] was also applied iteratively to select the most effective optimisation at reducing energy, one at a time [20]. A statistical test was used to choose

the optimisation which 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% over the highest optimisation level. This study used a larger range of optimisations than most, exploring 31 different optimisations.

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.

Milepost [21] used machine learning to select a set of optimisations to apply to a benchmark based on static features of the 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. [22] used features of the benchmark to predict what 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. To gather this data, hardware counters were used with the paper present which particular types of counters were found to be most informative. They were able to achieve a 17% improvement in performance with only 2 additional runs of the benchmark.

2.2.3 Ordering

Optimisation selection has been shown to both increase performance and decrease energy consumption to a moderate degree. However, by considering the ordering of optimisations much greater gains can be gained.

No 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 in [23]. 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, 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 [24] based on features from the application being compiled. This approach used Neuro-Evolution for Augmenting Topologies (NEAT) [25] 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.

2.2.4 Time-energy correlation

There have been many studies that look at the effect of optimisations on execution time [26, 27], and several studies suggesting that execution time can be used as a proxy for energy usage [28, 29].

The topic of performance and energy being highly correlated is addressed in [30]. This work explored several different overall optimisation levels, as well as four specific optimisations, using the *Wattch simulator* [31] to estimate energy results. However, the specific optimisations were all applied individually on top of the first optimisation level, without

exploring any possible interactions between the optimisations. The main conclusion drawn from this study was that most optimisations reduce the number of instructions executed, hence reducing energy consumption and execution time simultaneously.

Of the studies that look at individual optimisations and their effects on energy or power, most focus on only a few optimisations in isolation and few consider multiple platforms with different architectural features. Commonly explored optimisations, such as loop unrolling [32], loop fusion [33], function inlining [34] and instruction scheduling [35], have been examined extensively for different platforms using both simulators and hardware measurements.

2.2.5 Superoptimisation

Superoptimisation is a technique for producing perfectly optimal code, by doing a brute-force search of every possible instruction sequence. As this is impractical for longer sequences, ways to speed up this search have been developed and methods of quickly searching the space have been attempted.

Superoptimisation was first introduced in [36] where a superoptimiser was used to optimise short functions on a Motorola processor. This initial study was limited to only creating sequences consisting of register to register data processing instructions, with no memory accesses or branches. However, interesting and unexpected sequences were found that had not been considered before were found.

The instruction sequences were tested by running them with some input vectors on a Motorola processor. If the sequences passed enough test cases, it was considered correct, and then verified by hand to ensure it was correct for all possible inputs.

Source	**Naïve**	**Superoptimised**

`if(a < b)`
`  c++;`
	`  cmp eax, ebx`
	`  jge L1`
	`  add eax, #1`
	`L1:`
	`  ...`
		`subl eax, ebx`
		`adcl eax, #0`

Once the superoptimizer had found shorter sequences for many different types of these branches, the resulting sequences were inserted into GCC as peephole optimizations. This meant that the search did not need to be repeated during the compilation of any subsequent application.

A different approach to superoptimisation was taken by Denali [37]. This study attempted to use a solver with a set of rules to produce optimal programs. The rules specified each instruction's capabilities and the solver attempted to find the most optimal combination of instructions for a given function. However, few results were ever presented from this attempt, and the quality of the resulting sequence depended heavily on the quality and completeness of the rule set.

Gulwani et al. [38] present techniques for creating a program from a specification of how the program should behave. This technique is similar to the Denali superoptimizer, in that a set of logic rules describing components available for the synthesis and the functional specification is created, then given to a SMT solver to be solved in less than exponential time. This work focuses on creating loop-free constructs, due to difficulties in formulating the constraints for programs with loops. The process is semi-interactive, with the use providing input as to which components are to be available during the synthesis. This has the advantage of restricting the number of items needed to be considered, but limits the potential

for unique program formulations. By encoding the operations in this way, the system often produced good results, but made no guarantee of optimality.

The concept of reusing the discovered code sequences was extended by Bansal et al. [39]. This study harvested a much larger number of target sequences from a benchmark suite, then used a superoptimizer on all of these sequences. To avoid repeating the superoptimization process for each input sequence, the sequence was 'fingerprinted' by evaluating it for some specific test data. When the superoptimizer generated a instruction sequence, this code could be tested with the same input, and then its output matched to fingerprinted target sequences. By using this technique, the superoptimizer could be used for many different input codes simultaneously.

If matching sequences were found, these were then verified to be equivalent using a SMT solver and if this check passed, entered into an optimization database. This database was used to implement a peephole pass in a compiler and could be created offline (with up to instruction sequences of length 4) and then the peephole pass only needed to look up sequences in the database, greatly decreasing compile times.

Using this peephole pass with the generated optimization database resulted in code being iteratively improved as the pass was run multiple times, eventually achieving a performance similar to that when all other optimizations were enabled.

Stochastic superoptimisation is a technique explored in [40]. An intelligent hill climbing method was used to explore different instruction sequences, with feedback on how close the result was to the target directing the search. By exploring the space in this way, much longer instruction sequences (up to 16 instructions) could be considered. This lead to large performance increases being found over already heavily optimised code.

So far superoptimisation has not be considered for energy consumption — there is the possibility that if targeted for energy consumption many interesting and unexpectedly energy-efficient codes sequences could be found.

2.3 Energy modelling

Energy modelling is important because it allows software's energy consumption to be estimated without taking physical measurements of the hardware. This allows a faster design cycle and the ability to optimise for power without having the hardware to test on.

2.3.1 Low-level energy models

One of the first studies to consider energy consumption of software was performed by Tiwari et al. [41]. This created a model where each instruction was given a base energy cost, and each pair of instructions an inter-instruction effect. The inter-instruction effect was included as a result of bits in the processor's pipeline being switched as the next instruction was executed.

This allows the energy of a full program, p to be found:

$$E_p = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k, \tag{2.1}$$

where B_i is the base cost of instruction i, N_i is the number of times i is executed in p, $O_{i,j}$ is the inter-instruction cost from i to j, and $N_{i,j}$ is the number of times this interaction occurs in p. This model requires that energy figures for all instructions and pairs of instructions are found, which can be particularly challenging with some types of processor architecture. The final term in the model uses the parameter E_k to define 'other effects' not captured by

the model. This would include sources of energy consumption that are not directly linked to the currently executing instruction, such as periperhals and cache activity.

Tiwari's model is extended in [42] to model the instruction and data energy cost of the processor by considering the number of 'bit flips' between successive instructions. This model is significant because it takes into account the energy cost of the data the instructions are executing on. However, this model also suffers from the difficult of having many parameters which require difficult to measure empirical values.

Wattch [31] provides a generic framework for energy modelling, compiler optimisation testing and hardware design space exploration. This is enabled by modelling various hardware components at different levels, and allowing them to be combined into a full model of a processor. This processor is currently for non-superscalar processors, however other work has attempted to model superscalar processors [43]. The Wattch simulator is designed to allow easy energy measurements while exploring architectural configurations and is established at being within 10% of an industry layout-level power tool. However, Wattch does not model every hardware component in the processor, which makes it difficult to be certain about the total energy consumption of the processor.

SimplePower [44] is another simulator that has been used to explore the energy consumption of the software running on a processor. This simulator targets a five stage RISC pipeline, with energy consumption estimates based on the number of transitions on bus signal lines as well as various other components.

Various other models have been created to simulate power dissipation of the processor, including complex instruction level models [42], function-level models [45] and hybrids of these [46]. However, these all suffer the drawback that some energy consumption effects may not be modelled, potentially skewing the results.

3

Contributions

This section details published and unpublished work I have done over the last year.

3.1 Existing compiler optimisations

3.1.1 Optimisation analysis

This section discusses an analysis made of how optimisation selection affect energy consumption, across several different platforms and a set of benchmarks. As part of this study, a set of benchmarks was developed to expose the energy consumption characteristics of processors. This benchmark suite is detailed in Section 3.4.

The following hypotheses have been investigated, resulting in [A]:

1. The time and energy required for a computation are always proportional to one another. Examples were found where energy and time are not correlated and investigated.
2. There exists a set of compiler optimisations that gives a lower energy consumption than the predefined optimisation levels.
3. It is possible to search the compiler optimisation space in an efficient and systematic manner, to assign each optimisation an overall effectiveness.
4. There is no universally good optimisation across multiple benchmarks and platforms.

These hypotheses were explored for five different processors: ARM Cortex-M0, ARM Cortex-M3, ARM Cortex-A8, XMOS L1 and Adapteva Epiphany. This allows comparisons of the optimisations effectiveness across multiple different processor architectures and an exploration of how this affects energy consumption. The following contributions are made:

- The use of fractional factorial design to analyse a previously intractable optimisation space of GCC's optimisation options.
- Analysis of relative importance of each optimisation across multiple benchmarks and platforms.
- The answers to the previously given hypotheses.
- Commentary on how these techniques and results can be used by application developers and compiler writers.

Time and energy

The following section addresses the first hypothesis, and show that energy consumption and execution time are proportional to each other across all the benchmarks and platforms. A

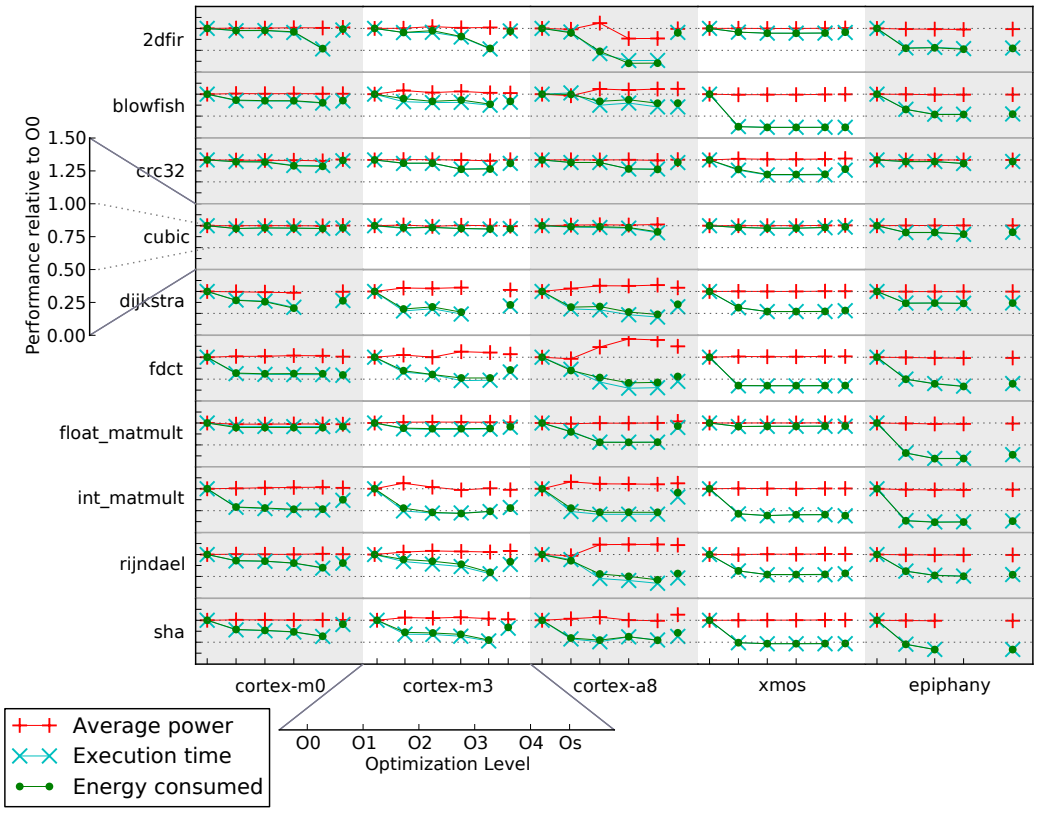


Figure 3.1: Energy, time and power results for benchmark-platform combinations. Optimisation levels `O0` to `O4`. `O4` is `O3` with link-time optimisation. The last point is `Os` — optimise for space. Some results are unavailable for when the compiler crashed while producing the output binary.

high level overview of each platform and benchmark for the different optimisation levels is given in Fig. 3.1. This figure shows a line graph for each combination, displaying the effect of the broad optimisation levels `O1`, `O2`, `O3`, `O4` (defined as `O3` with link time optimisation) and `Os` (optimise for space) on time, energy and average power when compared to the same program with all optimisations disabled.

For the Cortex-M0, very little difference between energy and time is seen due to it being the simplest processor tested, it has a three stage pipeline without forwarding logic. The pipeline behaviour is simple, only stalling if it encounters a load or a branch, thus it is not sensitive to specific code sequences. The Cortex-M3 exhibits very similar behaviour, with some very slight differences between energy and time. The micro-architecture in this processor is more complex, featuring branch speculation and a larger instruction set [47].

The XMOS processor has a four stage pipeline, similar to the Cortex-M3 in complexity and performance. It should also be noted that the compiler for the XMOS processor uses an LLVM backend [48] for code generation, featuring different optimisations. Due to this the result set for this processor is not as extensive as the other four, but is still broadly comparable.

The Epiphany processor also sees a large correlation between the energy consumption

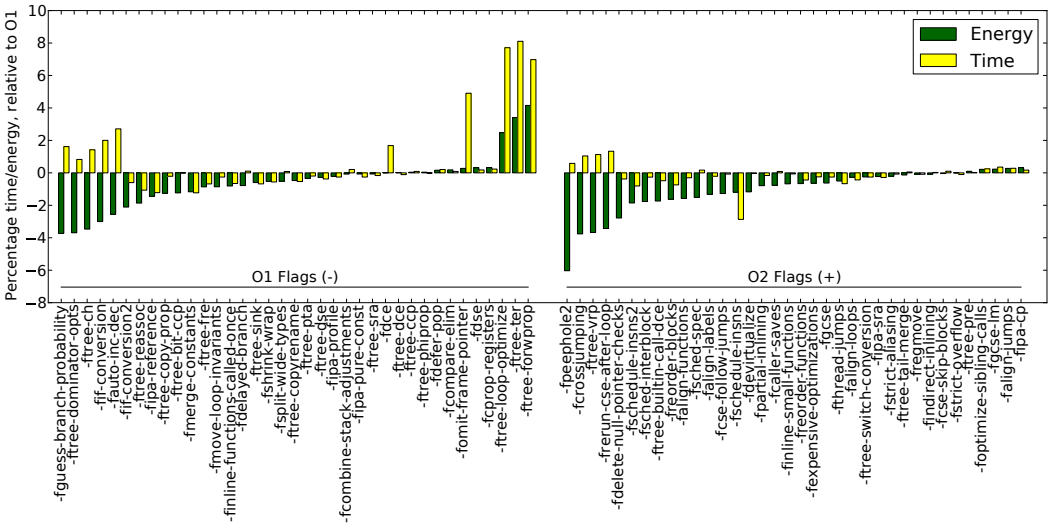


Figure 3.2: Blowfish benchmark on the Cortex-M3 platform. Individual options are enabled or disabled on top of the `O1` optimisation level.

and execution time. There is some divergence when the superscalar core in the processor is able to dispatch multiple instructions simultaneously. This gives the compiler more potential for creating advantageous code sequences.

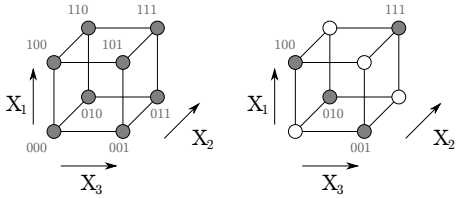
The greatest difference between energy and time was discovered while using the Cortex-A8. For the majority of the benchmarks the execution time reduces more than the energy. This is due to multiple instructions being executed simultaneously by the superscalar core, reducing the amount of time taken but not the energy consumption, as the same total work is still being done. We infer from this that the amount of pipeline activity has a significant measurable effect on the energy consumption. The gap is also seen to widen at the `O2` level, due to instruction scheduling being enabled there.

These results support our first hypothesis that time and energy are broadly correlated. The strongest correlation occurs in the qualitatively 'simplest' pipelines. Increasing pipeline complexity means there are more opportunities for architectural energy saving measures (clock gating, etc.) making the complex processor's energy profile more variable and improving the potential for compiler optimisation impact.

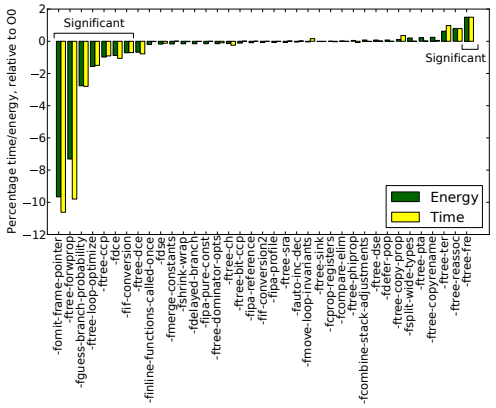
Optimisation potential

The second hypothesis to explore is that it was possible to find a set of optimisations that perform better than the standard optimisation levels. Fig. 3.2 shows each option in `O1` and `O2` optimisation levels enabled on top of the flags in `O1`. By examining the left of the graph, it can be seen that by disabling `-fguess-branch-probability` (in this specific run) the energy decreases by 4% at the expense of some additional run-time. This shows that a set of optimisations that performs better than the predefined `O1` optimisation level.

This conclusion is in line with much of the related work, that has focused on choosing a set of optimisations which is more optimal than the standard optimisation levels for a given benchmark.



(a) Reducing a 3-factor full factorial design to a 'half fraction' design.



(b) Blowfish benchmark on the Cortex-M0 platform. Individual options enabled at `O1` are listed.

Fractional factorial design

This section explores the third hypothesis — a method to systematically explore the optimisation space.

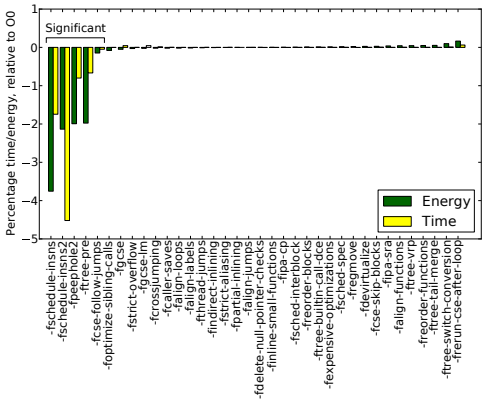
GCC has over 150 different options that can be enabled to control optimisations. The majority of these options are binary — the optimisation pass is either enabled or disabled. To further complicate matters, an optimisation path may be affected by other passes happening before it. It is not feasible to test all possible combinations of options, therefore a trade-off has to be made. One of our main contributions is to deploy fractional factorial design [19] (FFD) to massively reduce the number of tests to explore the space, whilst still identifying the options that contribute to run-time and energy. This approach has been explored on a small scale in [18], where nine optimisations were explored in just 35 tests as opposed to the 512 required for a full factorial design. It has also been explored by Haneda et al. in [26], where a fractional factorial design is used to inform the choice of optimizations. We apply this technique to allow us to analyse and draw conclusions about these large number of optimizations.

An example *full* factorial design is shown on the left of Fig. 3.3a. This example shows three factors with every possible combination enumerated. A fractional factorial design with the number of tests halved is shown on the right, yet still allows the difference between any two factors to be estimated.

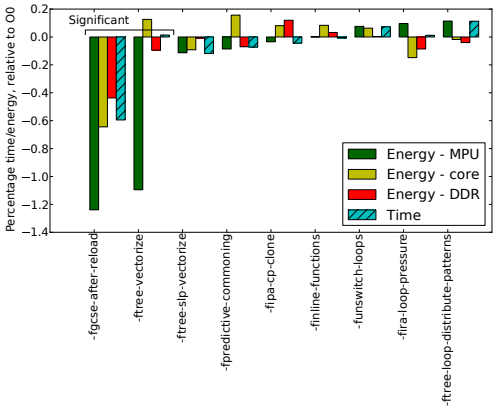
The drawback to this approach is that the high-order interactions between options (effects due to multiple options being enabled) will not be discernible. Fortunately, this is not usually a problem as these types of interactions are statistically rare. The degree to which this happens is specified by the FFD's resolution. A *resolution 5* design ensures that the main effects are not aliased with anything lower than 4th order interactions.

Using the Yates algorithm [19], the effect for any single or combination of factors can be found from the data. This gives an estimate for how much this factor or interaction affects the result of the experiment. The Mann-Whitney statistical test is used to determine whether the factor represents a significant change in performance as detailed in [20] and [49].

All FFDs used were generated by the statistical program, R [50] (a statistical programming language), using the FrF2 library [51].



(a) FDCT benchmark on the Cortex-M3 platform. Individual options enabled at `O2` are listed.



(b) 2D FIR benchmark on the Cortex-A8 platform. Individual options enabled at `O3` are listed.

FFD results

The results from the FFD experiments provide additional evidence to back up the first hypothesis, that execution time and energy are correlated.

Results showing the correlation between time and energy are shown in Fig. 3.3b. This shows the main effect each optimisation has on the runtime and energy consumption, as calculated by the FFD. A small percentage change is statistically significant because these results are derived from a total of 2048 separate runs. This significance is calculated using the Mann-Whitney test. The bracket above the bars indicates when the result satisfies the following hypothesis: there is 95% certainty that the result represents a significant impact on the energy consumption of the benchmark.

Fig. 3.4a highlights a discrepancy that occurred between execution time and energy consumption, even for very similar optimisations. The first two options listed (`-fschedule-insns` and `-fschedule-insns2`) both schedule instructions to reduce pipeline stalls. However the latter option performs its scheduling pass after register allocation, whereas the first performs it before. The option to schedule instructions after the register allocator can be explained by recognising that the scheduling will reduce stall cycles, which have a below average energy consumption. Overall, this reduces time more than energy (removing cycles that are below average energy will increase the average energy). The other option, however, is more unexpected in that the energy is reduced by a higher proportion than execution time. Upon further investigation this is partly due to fewer spill instructions being generated and partly due to instruction set effects. The scheduling allows causes some register-specific instructions to be converted to ones that are able to access additional registers, further removing the need to access memory.

Efficient SIMD Units An interesting effect is seen in 2D FIR for the Cortex-A8. The execution time decreases more than the energy consumption up to `O2`. However, when enabling `O3` the proportional decrease in energy is greater than execution time (a lower average power). On further investigation, this is caused by the `-ftree-vectorize` optimisation having an impact on energy consumption with no change in execution time (shown in Fig. 3.4b). This option vectorizes loops, so that SIMD instructions can be inserted. We do not see a performance boost due to the structure of the Cortex-A8 pipeline, where it is expensive to

NEON	Instruction Dependencies	Continuous Power Consumption
No	Yes	168 mW
No	No	195 mW
Yes	Yes	158 mW
Yes	No	159 mW

Table 3.1: Micro-benchmark results for multiplications on the NEON unit, with and without inter-instruction dependencies.

copy results between the NEON unit and the standard registers.

Further investigation of the NEON SIMD unit was done using some simple tests consisting of executing a single instruction many times. The results of these are shown in Tab. 3.1, showing doing continuous multiplication on the NEON unit uses around 20% less power than using the normal Cortex-A8 multiplier. When considering the similar number of cycles to execute each type of multiply, this results in a reduction in energy consumption when using the NEON unit. This is in line with what previous studies have found [11] and shows that by using the hardware to its full capacity, the greatest energy savings can be achieved.

Conclusions

The first hypothesis of energy consumption and execution time being correlated in the general case was found to be correct across many platforms and benchmarks. This was first shown to be true by the high level results, showing only the overall optimisation levels. The more detailed fractional factorial design runs also demonstrated this result, showing that most optimisations had the same relative effect on energy and time. This result occurs because the majority of optimisations focus on reducing the total amount of work performed by the benchmarks — thus minimising both energy consumption and execution time.

By adding and subtracting individual flags on top of the whole optimisation levels we have shown that a better set of flags exists, which can produce more optimal applications. This validates our second hypothesis, giving results in line with much previous work.

The third hypothesis stated that it was possible to efficiently search the optimisation space to gain information about the effectiveness of each optimisation. To perform this we leveraged fractional factorial designs, allowing us to test each optimisation in a greatly reduced number of runs. This method allowed us to explore complex effects seen on the Cortex-A8, where the SIMD unit helped achieve lower energy consumption.

The fourth hypothesis of there being no optimisation which was effective for all benchmarks and platforms was evaluated using fractional factorial designs (more detail available in the full paper [A]). We were able to extract the most effective optimisations for each benchmark and platform pair and these results showed that there was no single optimisation that was universally effective. Further analysis of adding and subtracting individual flags showed that the optimisation space is chaotic, with optimisations interacting in unpredictable ways.

The compiler writer can use these results and the fractional factorial design method to evaluate potential optimisation passes, ensuring that they perform well in a variety of configurations. Until a method for resolving the interactions between optimisations is found, it is envisioned that the developer could use this technique to eliminate optimisations that are not having a positive effect on their application. This will speed up compilation time as well as potentially improving the performance of their application.

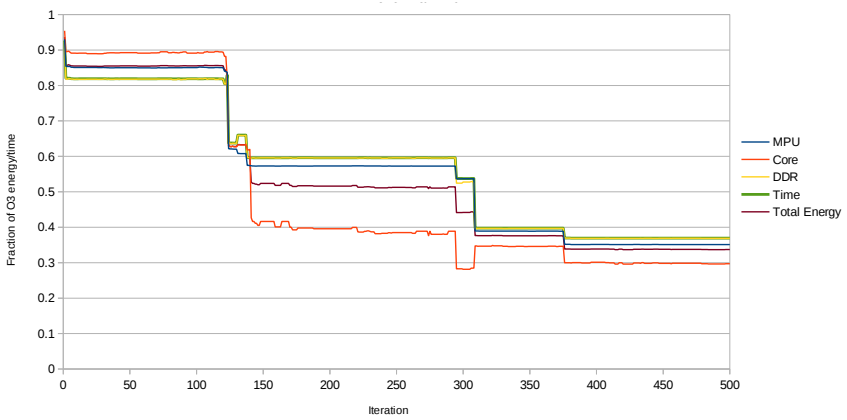


Figure 3.5: Genetic algorithm targeting the Cortex-A8. Overall energy is minimised, breakdown of energy into MPU, core and DDR memory is shown, along with time.

3.1.2 Optimisation selection

As seen in the previous section, the choice of optimisations can have a large effect on energy consumption. In this section the iterative elimination technique was performed on a variety of benchmarks [17]. This lead to a range of improvements between 0 and 11%. The algorithm was modified to retest flags which had been turned off, allowing it to consider a wider range of flags. This gave a small improvement over iterative elimination for some benchmarks, as it forming a type of hill-climbing algorithm. The results are shown below.

Benchmark	% improvement	
	Iterative elimination	Modified version
2dfir	8.3	8.3
blowfish	1.5	1.5
crc32	0.7	1.0
fdct	11.0	15.3
float matmult	1.9	2.1
int matmult	4.6	4.7
sha	0.3	0.3

3.1.3 Ordering of compiler optimisations

Some of the background work has been repeated, verifying their results. In particular, a genetic algorithm was ran on ordering the compiler optimisations. The results of this are shown in Figure 3.5. This result is in line with previous literature, showing that significant savings over the best optimisation level can be made by choosing and ordering the optimisations. This particular run of the genetic algorithm reduced the energy consumption to 33% of the O3 optimisation level and execution time to 37%.

The order of LLVM's transformations is shown to have a large effect on the energy consumption of a program in Figure 3.6. This graph shows how pairs of transformations change the energy consumption compared to no transformations. There is a large variation in energy consumption after just two transformations, from 8% increase to 12% decrease in

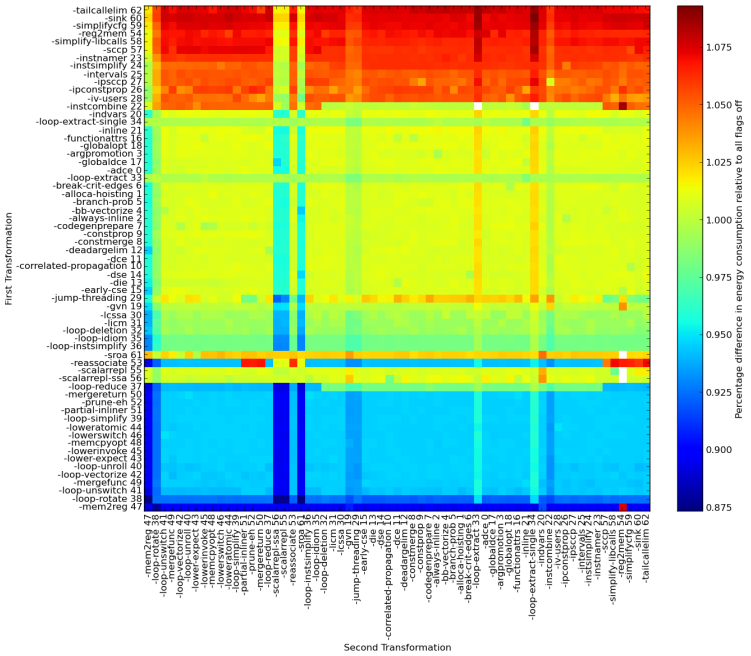
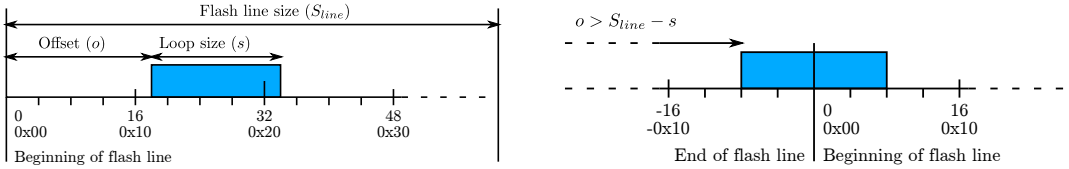


Figure 3.6: Pairs of transformations from the LLVM compiler, ran on the 2DFIR benchmark. Graph shows the change in energy compared to running no transformations.



(a) The offset and size of the loop relative to the beginning of the flash line.

(b) The loop crossing two flash lines.

Figure 3.7: Diagrams of how loop alignment can be tested.

energy consumption. The work is ongoing and will eventually lead to a similar analysis of ordering compile optimisations as was seen in Section 3.1.1.

3.2 Memory alignment

An optimisation for small embedded platforms, would typically overlook the alignment of code and data in memory, due to the fact that both RAM and flash are single cycle access across the memory space. However, the amount of energy consumed is not always equal, with interactions between the flash, the SoC and the processor. In particular, embedded flash memory is often divided into pages, lines and blocks, and changing the area of memory being accessed has an associated energy cost.

The energy required by different areas of memory was tested by choosing loops of different size and alignment, and measuring their energy consumption, as seen in Fig. 3.7a. In this

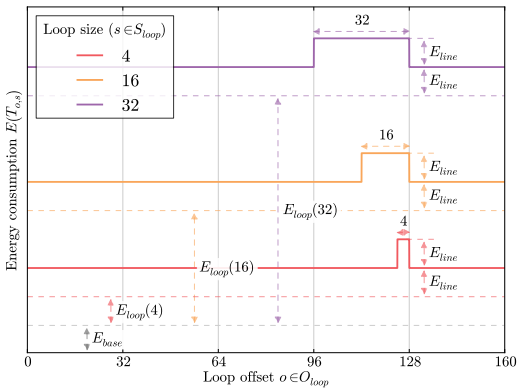


Figure 3.8: Energy consumption profile as given in Eq. 3.4 for $S_{loop} = \{4, 16, 32\}$ and $S_{line} = 128$.

diagram, S_{line} is the size of the flash line, o is the offset of the loop in memory and s is the size of the loop.

In the tests run, $T_{o,s}$, where $o \in O_{loop}$ and $s \in S_{loop}$:

$$O_{loop} = \{x | x = 0, 2, 4, .., 256\} \tag{3.1}$$

$$S_{loop} = \{x | x = 8, 10, 12, .., 128\}\,. \tag{3.2}$$

Therefore the range of tests covered is given by:

$$T_{o,s} = O_{loop} \times S_{loop} \tag{3.3}$$

From the hypothesis that powering up a single line requires E_{line} joules, the energy required by $T_{o,s}$ is given below.

$$E\left(T_{o,s}\right) = E_{base} + E_{loop}(s) + E_{line} \cdot \left\lceil \frac{(o \bmod S_{line}) + s}{S_{line}} \right\rceil \tag{3.4}$$

E_{base} is the base energy required, and $E_{loop}(s)$ is the energy that would be required by the instructions in the loop with no flash lines powered up. This idealised model is shown in Figure 3.8. This diagram shows how each energy value in the previous equation is likely to appear when plotted.

The effect on energy of these characteristics can be seen in Fig. 3.9. This graph shows the data as recorded above for several platforms. Interesting features are annotated and discussed below.

A This is the effect of powering up an additional flash line, as predicted by the structure of the underlying flash memory. This manifests as a spike in energy consumption when the loop spans two flash lines (as shown in Fig. 3.7b). This happens when:

$$o \bmod S_{loop} > S_{line} - s \tag{3.5}$$

These spikes are not seen on the PIC32MX5XX, as the prefetch cache masks the energy consumption by having the cache line preloaded.

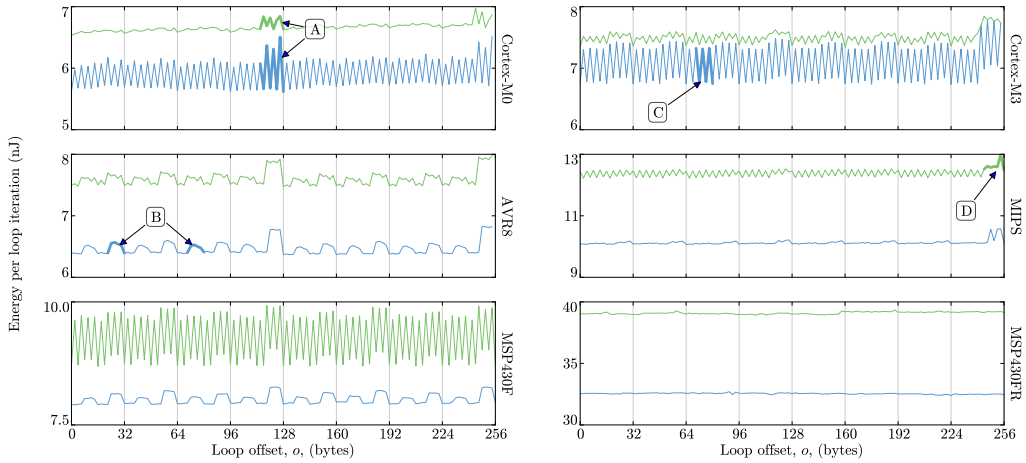


Figure 3.9: The effect of loop alignment on energy consumption, for $S_loop = 8, 10$.

B Increases in energy consumption are seen when the loop starts in the last 8 bytes of a 16 byte block. This is seen on AVR8 on the Cortex-M3 to a lesser extent. This is also a artefact of the flash structure: flash cells are grouped into blocks of 16 (AVR8) or 32 bytes (Cortex-M3). When the loop straddles multiple blocks, addition energy is required to activate both blocks simultaneously.

C On the Cortex-M0 and Cortex-M3 platforms the alignment to a 4-byte boundary has a large effect on the energy consumption. This effect occurs because the flash cell size is 4-bytes.

D The MIPS and Cortex-M3 platforms don't have large spikes at 128 bytes, but do at 256 bytes, suggesting that their flash line is 256 bytes long.

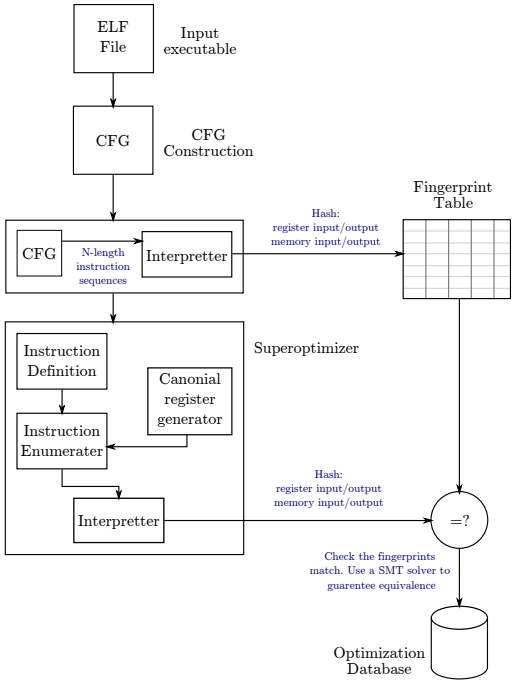
Item **B** discussed the effect of blocks in flash. This can be incorporated into the previous equation by adding another term. The E_{block} parameter is the amount of energy required to enable a flash block and S_{block} is the size of the flash block.

$$E\left(T_{o,s}\right) = E_{base} + E_{loop}(s) + E_{line} \cdot \left\lceil \frac{(o \bmod S_{line}) + s}{S_{line}} \right\rceil + E_{block} \cdot \left\lceil \frac{(o \bmod S_{block}) + s}{S_{block}} \right\rceil \tag{3.6}$$

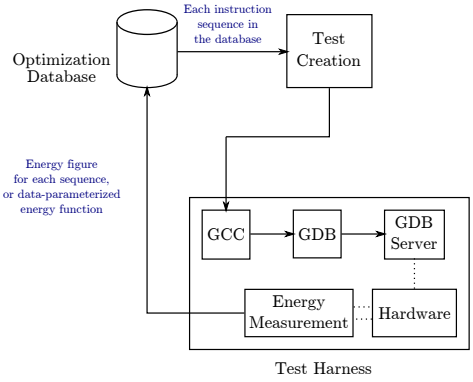
There are a variety of effects seen across these platforms, work is continuing to explore these and the interactions between processor architecture, SoC design and memory technology. These effects could be exploited by a compiler optimisation to make sure that loops and frequently executed areas of code do not cross these memory alignment boundaries. This should reduce energy consumption in devices with embedded flash.

3.3 Superoptimisation

The section of work describes an ongoing attempt to create a superoptimiser targeting energy consumption. The aim is to try and find the best possible sequence of code in terms of energy consumption. A methodology similar to Bansal et al.[39] is followed, by finding equivalent



(a) First stage of the database creation process.



(b) Second stage of the database creation process: measuring the energy of each sequence.

sequences of code, and them applying them to executables. This first requires a database of equivalent instruction sequences is created.

The creation of the optimization database is split into two parts:

1. Finding functionally equivalent instruction sequences to the target sequence. This phase harvests instruction sequences from an executable and inserts them into a fingerprint hash table. The superoptimizer is then run, matching sequences against this hash table. If the sequences are verified to be correct then they are entered into the optimization database. This is shown in Figure 3.10a.

2. Costing the instruction sequences. This phase costs each instruction sequence in the database by running it on hardware under a variety of test cases. This allows an energy figure or an energy function, based on the data to be found. This phase is shown in Figure 3.10b.

Once this optimization database has been created, it can be used in a peephole pass when compiling the final application. This is simply a look up into the optimization database, replacing the target sequence with the best found.

The optimization database stores all found sequences, along with their performance parameters. By storing several parameters, such as energy, execution time and sequence size, trade-offs can be made when compiling a target application. The data points needed to be stored by the optimization database are shown in Table 3.2.

As the database stores several different cost metrics, the resulting program can be balanced between the three metrics. This allows maximal performance for a given energy level, for example. It also allows the Pareto frontier to be calculated, highlighting which optimizations result in the best trade-off between code size, performance and energy consumption.

Field Name	Description
Input Sequence	The input sequence is the sequence that was harvested from during the training process.
Replacement	This is the sequence being scribed in this record of the database — the sequence found by the superoptimizer to be equivalent to the input sequence.
Energy cost/function	This metric records the energy consumption of the replacement sequence, relative to the input sequence.
Execution time	This records the execution time, relative to the input sequence.
Code size	This metric holds the code size of the replacement sequence.

Table 3.2: Fields required in the superoptimisation sequence database.

3.3.1 Pruning the search space

The size of the search space is given by the formula below:

$$S = |I|^l \tag{3.7}$$

where I is a subset of the instruction set and l is the length of the instruction sequence to be generated.

It can be seen that the size of the space grows exponentially with l, therefore the complexity of the functions that a superoptimizer is able to generate/ and the length of time required to find a correct function is directly related to this search size. A search space which has not been pruned results in an extremely long number of instruction sequences that need to be checked before a correct one is found.

This section discusses several techniques that can be used to decrease the size of the search space and their implications on the type of sequences that can be found. The pruning techniques must be selected carefully: choosing a complex technique could greatly lower the size of the search space, however if this is a costly technique to implement the overall time to search the remaining space may be larger.

Canonical form

This technique was first introduced by Bansal et al. [39] exploiting the orthogonality of registers in the instruction set. The instruction sequence is reduced to a unique form by renaming the registers. The following table shows several single three-operand instructions transformed into canonical form. All of the registers first appear in ascending order in the transformed sequences.

`add r1, r0, r0`	$\longrightarrow$	`add r0, r1, r1`
`add r4, r0, r3`	$\longrightarrow$	`add r0, r1, r2`
`add r0, r5, r2`	$\longrightarrow$	`add r0, r1, r2`

Transforming the instructions in this way greatly reduces the number of possibilities there are to examine. For a single three-register instruction, assuming 16 registers, the search space reduces from $16^3 = 4096$ to 5 unique combinations. These combinations are given below:

input : L = list of previous numbers, initially $(0, 0, ...)$
input : k = maximum number of registers
input : $n = |L|$
output: L
output: Returns true if exhausted entire sequence.

```
for i ← 0 to n − 1 do
    if L_i < max(L_{i+1}...L_n) + 1 and L_i < k then
        L_i = L_i + 1 ;
        return false ;
    else
        L_i = 0 ;
    end
end
return true ;
```

Algorithm 1: Algorithm to return the next register renaming.

```
add r0, r0, r0
add r0, r0, r1
add r0, r1, r0
add r0, r1, r1
add r0, r1, r2
```

This same technique can be applied to instruction sequences by ensuring that registers are renamed so the registers first appear in ascending order.

```
mov r2, #1                    mov r0, #1
add r5, r5, r2                add r1, r1, r0
sub r2, r3, r5      ⟶        sub r0, r2, r1
shl r1, r2, #1                shl r3, r0, #1
add r1, r1, #1                add r3, r3, #1
```

Renaming the registers in this way greatly reduces the number of unique instruction combinations to examine. This renaming is equivalent to partitioning n elements into k subsets, where n is the number of register 'slots' and k is the number of unique registers to place in those slots. The number of possible combinations for a given n and k is given by the Stirling numbers of the second kind [52]:

$$S(n,k) = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n \tag{3.8}$$

The register renamings can be generated iteratively, from the previous renaming by using the algorithm 'Restricted growth strings in lexicographic order' listed in [53]. This algorithm is shown in Algorithm 1.

Using canonical form is valid if the registers are all orthogonal, which may not always be the case for energy. If the energy cost for accessing a specific register is high for a particular register then this transformation will mean that the most efficient instruction sequence may not be found. Renaming the registers will change the number of bit-flips, potentially affecting the energy. This effect is likely to be small, however, as previous work has determined these effects to be at least an order of magnitude smaller than other effects [54].

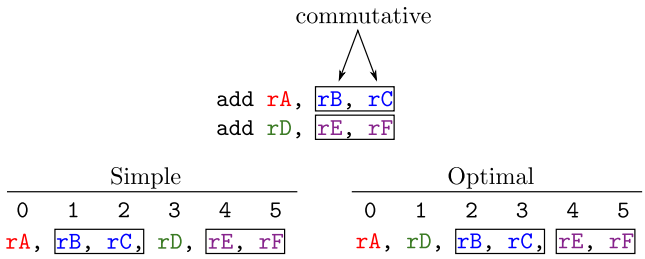


Figure 3.11: Commutative register arrangement for simple and optimal forms.

Accessing registers causes a temperature rise in the CPU, which can also affect energy consumption [55]. This renaming technique has the possibility of affecting the temperature of the registers, changing their energy consumption characteristics.

Commutativity

Another possible technique to reduce the number of register combinations to explore for an instruction sequence is by exploiting the commutativity of certain operations. The following instruction are functionally equivalent:

$$\texttt{add r0, r1, r2} \longleftrightarrow \texttt{add r0, r2, r1}$$

By exploiting the fact that certain instructions have this property the search space can be reduced. This effect is increased when combined with canonical form. Commutativity can be combined with canonical form by only accepting register sequences which have the commutative pairs in ascending order:

Invalid	Valid
`add r0, r1, r0`	`add r0, r0, r1`
`add r0, r2, r1`	`add r0, r1, r2`
`add r0, r1, r0`	`add r0, r0, r1`
`add r2, r3, r0`	`add r2, r0, r3`

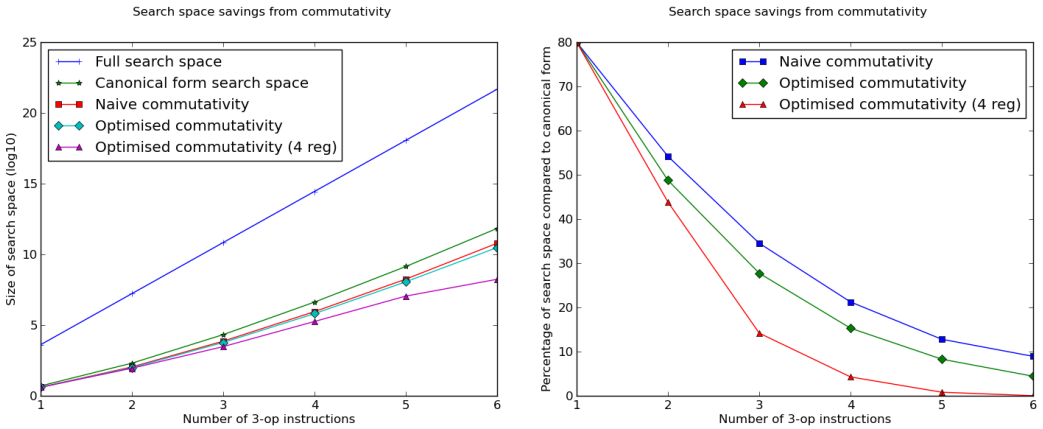
When canonical form is used, the reduction is search space size is dependent on the positions of the commutative registers — $L_i > L_j$ occurs more frequently for large i and j (in Algorithm 1), as the renaming ensures that registers are first used in ascending order. The maximum reduction in search space can be achieved by placing the commutative registers in the last positions. This difference between simple and optimal placement of commutative register slots is demonstrated in Figure 3.11.

This diagram shows that to achieve the maximum search space reduction the commutative register pairs need to be placed in this highest register slots. A simple method of using the registers in their defined order will not give the best results. The sizes of the search spaces are given below, in Table 3.3.

These numbers are graphed in Figures 3.12a and 3.12b. The first graph shows the overall search spaces for three-operand instructions without any reduction, using canonical form,

N	Canonical form	Simple	Optimal
1	5	4	4
2	203	110	99
3	21,147	7,301	5,865
4	4,213,597	894,904	644,417
5	1,382,958,545	176,869,540	114,514,772

Table 3.3: Size of the search space for instruction sequence length N of a single type of instruction with various reduction strategies.



(a) Size of search space

(b) Percentage of full search space

Figure 3.12: These graphs show how the size of the search space changes with increasing length instruction sequences for various method of reduction.

and then two possible reductions exploiting commutativity. The second graph shows the size of the search space after commutativity reductions, compared to canonical form.

From these graphs it can be seen that a significant reduction is given only using instruction sequences in canonical form, with small additional reductions by adding commutativity on top of this.

The selection of commutative instructions must be done with care, as there are often subtle restrictions on the operands that may be taken. One example may be where there is a subset of registers that may be accepted by one operand. Swapping the registers also has some similar impacts as canonical form — there could be a different pattern of switching behaviour which would affect the energy consumption.

This techniques relies on operations being truly commutative. Some operations, such as IEEE 754 floating point addition and multiplication are not necessarily commutative. The effectiveness of applying these rules is then limited by the number of operations available. This could be as few as just the addition, multiplication and several bitwise operators.

Dead code

This technique involves analysing the generated instruction sequence for dependencies that would have been eliminated in fully optimised code.

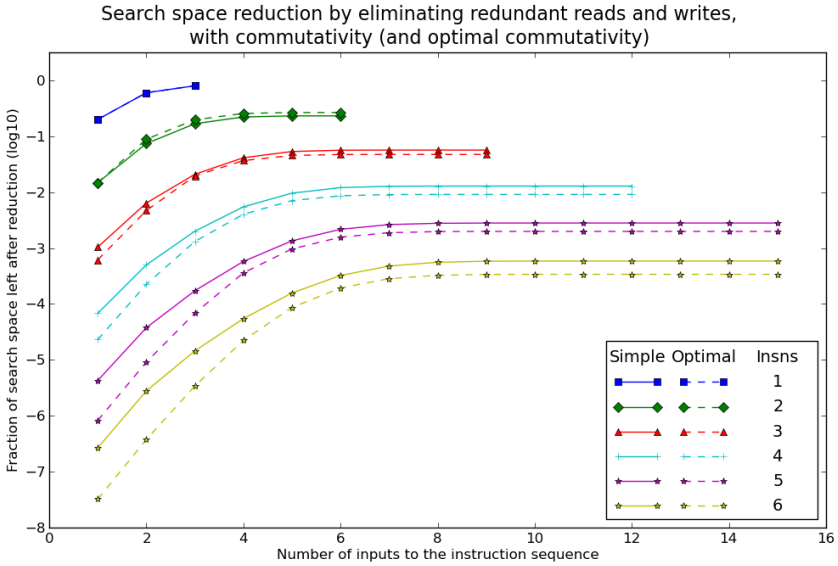


Figure 3.13: Reduction in search space achievable with dead code elimination.

Read from uninitialised register.
The number of registers in use, and used by the input sequence is known, therefore if the sequence reads from a register that has not been assigned to yet, it can be excluded.

Dead code elimination.
This technique removes code that is never used by the output. The instruction sequences is excluded from the search if it produces a result which is never read in the sequence, or required as an output from the sequence. As the instruction sequence produces a redundant result, an optimal instruction sequence would not have this instruction.

Filtering the instruction sequences in this way makes the size of the search space dependent on the number of inputs and outputs required by the target function. In the following example, the line highlighted contains a write to `r1` which is never used before being overwritten, so could be considered a non-optimal sequence.

Input: `r0`, `r2`, Output: `r0`

```
add r1, r0, #0x10
sub r1, r2, r0
and r0, r1, r2
```

Similarly, the code sequence can be considered non-optimal if the sequence produces an output that is not required. In the code below, the last statement writes to `r1`, which is not required as an output of this sequence.

Input: `r0`, `r2`, Output: `r0`

```
add r1, r0, #0x10
```

```
sub r0, r2, r1
and r1, r0, r2
```

A sequence can also be considered invalid if it uses a registers which is not live at the point of entry to the code. This can be found from liveness analysis of the surrounding code. In the code below, the register `r3` is not live on entry to this section of code, so its use makes the sequence invalid.

Input: `r0`, `r1`, Output: `r0`

```
add r0, r0, #0x10
sub r0, r3, r1
and r0, r1, r0
```

The amount of the search space that can be saved by exploiting deadcode is shown in Fig 3.13. This graph shows the dependency on number of inputs to the instruction sequence.

3.4 Benchmarking energy consumption

This section discusses a benchmark suite designed to measure energy consumption. This works is available as a preprint [B].

Benchmarking is frequently used to gain an idea of how a system will perform during general use, when the specific environment cannot be reproduced at design-time. This gives designers feedback on how their system will perform and where performance is lacking. Typically, one benchmark cannot exercise all aspects of a target, leading to suites of benchmarks. Each benchmark tests a combination of areas of the hardware. This separation of benchmarks allows the designer to see which parts of the hardware perform the best.

There are few freely available benchmark suites for deeply embedded systems and none exist which are designed to allow energy consumption to be measured. Existing suites, such as MiBench [56], MediaBench [57], LINPACK [58] and Dhrystone [59] are all targeted towards larger desktop-based applications, with significant compute power. This is due to their emphasis on measuring performance, as opposed to energy efficiency. Most assume a host operating system is present, which may not be true 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. There are a variety of issues that cause these difficulties, such as lack of an OS, lack of a storage system, small memory size and run-time scalability. The issue of run-time scalability only occurs with a diverse range of platforms — large differences in clock speed and microarchitecture may mean that without scaling down a benchmark it is infeasible to run it on less powerful platforms.

A new set of benchmarks has been created — the Bristol Energy Efficiency Benchmark Suite (BEEBS) [C] — modified from popular benchmark suites, and their use justified for benchmarking energy consumption. The benchmark suite is designed to expose the processor and memory's performance, with other factors such as I/O and peripherals excluded for portability. The selection was designed such that the benchmarks would be portable, to expose the changing in energy consumption when exercising the platform in different ways, such as with memory verses arithmetic intensive computation. The benchmarks are intended to be run on the bare metal with no host operating system.

Name	Source	B	M	I	FP	License	Category
Blowfish	MiBench	L	M	H	L	GPL	Security
CRC32	MiBench	M	L	H	L	GPL	Network, telecomm
Cubic root solver	MiBench	L	M	H	L	GPL	Automotive
Dijkstra	MiBench	M	L	H	L	GPL	Network
FDCT	WCET	H	H	L	H	None†	Consumer
Float Matmult	WCET	M	H	M	M	None†	Automotive, consumer
Integer Matmult	WCET	M	M	H	L	None†	Automotive
Rjindael	MiBench	H	L	M	L	GPL	Security
SHA	MiBench	H	M	M	L	GPL	Network, security
2D FIR	DSPstone	H	M	L	H	None†	Automotive, consumer

Table 3.4: Benchmarks selected, and the categories they fit in. Legend in Table 3.5.
† Redistributed under the GPL.

Key	Description
L	Low
M	Medium
H	High
B	Branching
M	Memory intensity
I	Integer pipeline intensity
FP	FPU pipeline intensity

Table 3.5: Legend for the benchmark table

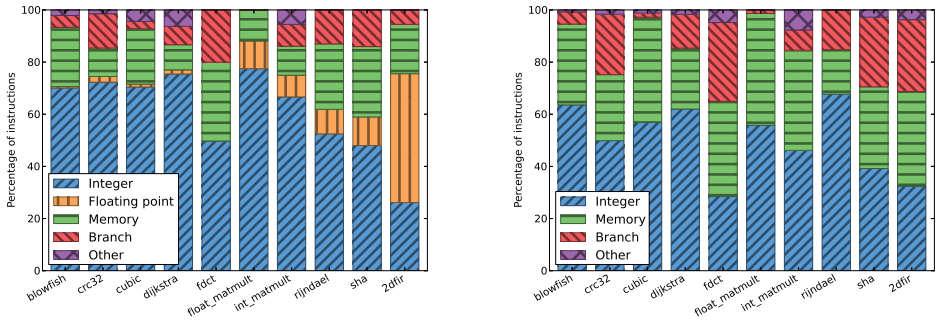
3.4.1 Benchmark selection

A set of benchmarks to tests all aspects of the target platforms is presented in this section. The benchmarks were selected by defining a coverage matrix which included all the individual benchmarks from following suites:

- MiBench
- DSPstone
- WCET
- Livermore Fortan Kernels
- Dhrystone
- MediaBench

The matrix (listed in full in Appendix A of [B]) also broadly evaluated other benchmark suites for their suitability. Two sets of parameters are evaluated in this table — type of operations performed by the benchmark and suitability for inclusion in the final suite. The suitability for inclusion evaluates whether the benchmark should be included, based on what the benchmark does, whether it will work on the target platforms and the effort required to port it.

The type of operations was derived from examining the source of each benchmark and roughly categorising it as to the types of operations it performs. This allows benchmarks with similar properties to be excluded before a lengthy examination.



(a) BEEBS Instruction distribution for the Epiphany platform.

(b) BEEBS Instruction distribution for the XMOS platform.

Benchmarks with a high suitability and a minimal set covering suitably different types of operations were selected to be included in the final suite (shown in Table 3.4). The types of operations are listed were calculated from a combination of inspecting the source code and from the instruction traces generated. This is shown in the table under the following columns:

- **B**ranching.
- **M**emory.
- **I**nteger.
- **F**loating **P**oint.

The final list of chosen benchmarks is shown in Table 3.4.

3.4.2 Benchmark analysis

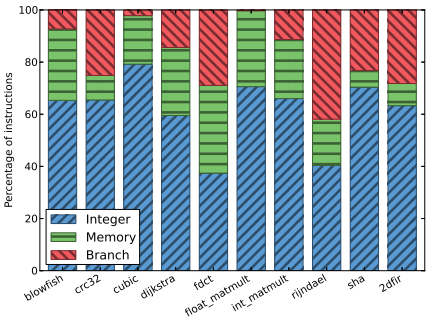
This section provides a concrete analysis of all the chosen benchmarks by collecting their instruction traces across three of the platforms. From these graphs, the instructions can be categorised to demonstrate that each benchmark performed a different distribution of operations. Figures 3.14a, 3.14b and 3.15a show the instruction distributions for the Epiphany, XMOS and ARM Cortex-M0 (Thumb instruction set) platforms respectively. The 'Other' category of instructions contains miscellaneous control instructions that do not fit into other categories (for example, interrupt control on the Epiphany platform).

Overall these results show that the benchmarks give a good spread of different distributions of instruction types.

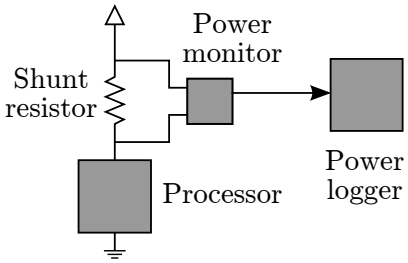
Integer operations are the most common type of instruction in almost every benchmark. Across the platforms, the distributions are similar, with small variations due to the underlying instruction set. For example, there are a larger percentage of `mov`-type instructions in the Epiphany results because there are several predicated `mov` instructions (`moveq`, `movlt`, etc). This reduces the need for conditional branches, so this category decreases in proportion.

Epiphany is also the only platform in the subset chosen which has hardware support for floating point. For the other platforms, software emulation is used. On the XMOS platform this manifests in extra branch and memory instructions, whereas for the ARM platform the proportion of integer operations rises. These differences are due to different emulation strategies used.

The ARM traces follow the same general trend as the traces for XMOS and Epiphany, however with overall less memory operations. This is due to the ARM processor having



(a) BEEBS Instruction distribution for the ARM Cortex-M0 platform.



(b) Hardware setup to measure the power of the processor under test.

Type	Platforms (%)	Benchmarks (%)		
		Epiphany	XMOS	ARM
I	30	26–77	28–68	37–79
FP	–	0–49	–	–
M	30	10–30	17–43	6–34
B	29	1–20	1–30	1–42

Table 3.6: Variation in instruction distributions between the platforms and between the benchmarks.

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.

The integer instruction category is the largest group in almost every case, for all platforms and benchmarks. This comes from the integer category covering the largest number of types of instructions, as it groups arithmetic, register copying and bit-wise operations.

These benchmarks show a range of different quantities of each instruction, with similarities across platforms. This makes the set of benchmarks ideal for use in energy profiling of a system.

We see that for all platforms a given benchmark produces a similar instruction profile (within 30% between all platforms). This is shown in Table 3.6, where the platforms column shows the maximum variation between each platform for each instruction category. The benchmark columns show the ranges of instruction proportions across the benchmarks on that platform. Between benchmarks there is significant variation, therefore the suite explores a wide range of input configurations in a consistent way between architectures.

3.4.3 Case study

The use of the benchmark suite is demonstrated through collecting power measurements for each benchmark on each of the platforms. Linear regression is then used to assign an average power dissipation to each class of instructions by considering the average power and instruction distribution per benchmark.

The power of each platform was measured by instrumenting hardware as in Figure 3.15b. This set-up allowed real measurements to be taken, rather than using an abstract power

Category	Power (mW) Epiphany	XMOS	ARM
Integer	28	33	8.4
Floating Point	31	–	–
Memory	20	35	9.3
Branching	40	35	6.8
Other	14	–	–
Average	26	35	8.3
Average/MHz	65μW	88μW	170μW

Table 3.7: Power dissipation for each instruction category calculated by linear regression.

model for the processor.

The average power dissipation of each benchmark was measured on the three hardware platforms. Linear regression is applied, with the categorized instruction counts gathered from the traces. This allows each category of instructions to be assigned an average power dissipation. The results of this analysis are presented in Table 3.7. These are scaled results, representing the cost of a single instruction per core/hardware thread (Scaled down by 16 for Epiphany and by 4 for XMOS).

Overall, the main difference in power dissipations is due to differing clock rates — XMOS and Epiphany run at 400MHz and ARM at 48MHz.

From these results several conclusions can be drawn. For the ARM Cortex-M0, a memory access is more costly than an arithmetic instruction, as is expected. The branch power dissipation, disagrees with other results taken. The power measured when executing a `while(1);` loop was found to be 11mW. This figure is higher than a memory access, due to the instruction being loaded from flash as opposed to RAM. The discrepancy is due to conditional branches having a lower power when the branch is not taken (further results indicate that when a conditional branch is not taken, the power dissipation is roughly 4mW).

The XMOS results show memory operations are slightly more costly than arithmetic. The identical cost for branching and memory access is due to the structure of the processor's pipeline: the final stage is a memory access which either does an instruction fetch or a memory operation.

The results for the Epiphany exhibit the most variability, with a branch instruction requiring almost twice the power of a memory access. We believe this is due to the longer pipeline having to be flushed, then new instructions fetched. A floating point operation also takes more power than an integer instruction — this is attributed to the larger complexity of an FPU.

4

Future work

4.1 Loop alignment in embedded devices

Many embedded devices use embedded flash as their non-volatile memory storage. The structure of flash in these devices has not be exploited to reduces the energy consumption of applications in most cases. In particular, when the program has hot loops which are incorrectly aligned the energy consumption is up to 15% higher than when the loops are correctly aligned.

Initial analysis over a range of platforms suggests that the savings are between 5 and 15% for aligned vs. unaligned loops (Section 3.2).

4.2 Effect of different memory technologies

It is cheaper to execute out of RAM instead of flash. Most of the current work in scratch pad memory has excluded deeply embedded devices, because the time to access both flash and RAM is single cycle. This has led previous studies to target larger memory hierarchies, considering performance rather than energy cost. Therefore, can we analyse the loops in a program and automatically copy the hot areas of loops to RAM, as a compilation post process?

4.3 Vector unit exploration

As seen in previous work, using the vector unit is more efficient than other functional units in the processor. Explore whether this is the case in other platforms, and whether it can be exploited by compilers to reduce energy consumption.

4.4 Other ideas

This section lists other potential ideas or topics that could be explored.

Hardware flash shadowing As it is costly to execute out of flash, shadow specified parts of flash in an execution buffer. The shadowing would be specified by an instruction, and instructions would be executed from this buffer. This allows loops to be held in more energy efficient RAM, instead of executed out of memory.

Optimisation ordering Can the optimisation ordering space be explored with a technique similar to fractional factorial designs?

Energy data modelling How can a model account for potentially significant changes in energy consumption based on the data being used by the instructions? How can we model this without knowing exactly what data will be used by a program?

Cache energy modelling Can a cache energy model be constructed from empirical energy measurements?

Superoptimization Does superoptimization for energy consumption produce different optimal sequences than if targeting code size or performance? If different code sequences are found, how different are they? Is it algorithmically different to the optimal code for performance? Is the optimal sequence dependent on the input data? Do different algorithms arise from different data?

Polling vs. interrupts Under what conditions is an interrupt driven system more energy efficient than a polling based one. Are there cases where polling uses less energy?

JIT for energy efficiency Is it possible to use Just In Time compilation on embedded platforms, to dynamically optimise for energy consumption?

5

Conclusion

This report has covered a variety of techniques that the compiler can implement to reduce energy consumption. Key to testing all of these was the development of a benchmark suite (BEEBS, Section 3.4) designed to expose the energy consumption characteristics of the processor it was running on. This benchmark suite consisted of 10 benchmarks, each carefully selected to test a wide range of behaviour on the target system.

An analysis of existing compiler optimisations was carried out (Section 3.1.1). This found that for most existing optimisations, energy consumption was improved in proportion to execution time, mainly as a result of fewer instruction being executed. However, it was also noted that all of the optimisations examined were created for performance reasons, with their energy benefit being a side effect. It was shown that the default set of optimisations could be improved upon, and that in many cases there were optimisations that had a detrimental effect on the energy consumption.

In some cases an optimisation was found to improve the energy consumption more than the execution time. In the two cases that were investigated, one cause was the reduction in memory operations (resulting in lower energy consumption, as predicted by the related work). The other case resulted from the multiplier in the NEON vector unit being more energy efficient than the Cortex-A8 core's multiplier and the optimisation made use of this functional unit.

The analysis concluded that there was no single set of optimisations which could be applied to all platforms and all benchmarks to result in good performance. This highlights the need for the compiler to be smart about the optimisations it chooses to apply.

Preliminary work was carried out with the ordering of compiler optimisations. Initial results suggest that the ordering of compiler optimisations can provide an even greater benefit than just optimisation selection. However, it is much more challenging to explore the optimisation ordering space, due to its large and unbounded nature.

A previously unconsidered effect from executed code out of flash was identified in Section 3.2. This explored devices with embedded flash, noting that when loops crossed certain boundaries in memory the energy usage increased. This is due to the structure of flash memory, and the need to power up additional bits of circuitry. This has the potential to be developed into an energy saving optimisations. Other optimisations to reduce energy consumption are listed in the future work section (Section 4). These include intelligently copying code into RAM for lower energy consumption and exploiting the vector units available in the processor.

Superoptimisation was explored as a possibility for generating new optimisations (Section 2.2.5). The superoptimiser harvests target instruction sequences from a set of benchmarks, and fingerprints them. Instruction sequences are then generated iteratively and

checked for functionality matches against the harvested sequences. If they match, their energy consumption is measured, and the pair of sequences is added to a database. This database can then be used by a compiler to implement a peephole pass, choosing the most energy efficient sequence of instructions. There is a very large space of instructions to search, and techniques to prune this space have been developed.

All of these techniques will individually increase the energy efficiency of software running on an embedded platform. However, it is unclear whether they will still be as effective when combined together, and this is an avenue for further exploration. A method of evaluating how frequently new compiler optimisations are effective was proposed in Section 3.1.1. By using fractional factorial design to explore how a new compiler optimisation interacts with existing ones, informed decisions can be made about which circumstances to enable the optimisation.

Overall, the techniques discussed in this report represent a first step towards giving the compiler the tools necessary to make a large scale effect the energy consumption of its target. These methods will be developed further over the course of my PhD with the goal of uniting them together. Each optimisation has a small effect when considered individually, but when combined it is expected that a large impact on energy consumption could be made.

Own Publications

[A] James Pallister, Simon J. Hollis, and Jeremy Bennett. "Identifying Compiler Options to Minimise Energy Consumption for Embedded Platforms". In: *Computer Journal* (2013).

[B] James Pallister, Simon Hollis, and Jeremy Bennett. "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms". In: (2013). arXiv:1308.5174.

[C] James Pallister, Simon Hollis, and Jeremy Bennett. *The BEEBS Benchmark Suite*. 2013. URL: http://www.cs.bris.ac.uk/Research/Micro/beebs.jsp.

See Appendix B of a list of presentations given and other activities.

References

[1] M. R. Stan and W. P. Burleson. "Bus-invert coding for low-power I/O". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 3.1 (Mar. 1995), pp. 49–58.

[2] Seungdo Woo, Jungmin Yoon, and Jihong Kim. "Low-power instruction encoding techniques". In: *SOC Design Conference* (2001).

[3] Peter Marwedel et al. "Fast, predictable and low energy memory references through architecture-aware compilation". In: *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No.04EX753)* 1 (2004), pp. 4–11.

[4] Stefan Steinke, Lars Wehmeyer, and Peter 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.

[5] Lovic Gauthier et al. "Minimizing Inter-Task Interferences in Scratch-Pad Memory Usage for Reducing the Energy Consumption of Multi-Task Systems Categories and Subject Descriptors". In: *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*. 2010.

[6] Xuan Guan and Yunsi 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.

[7] K Asanovic. "Energy-exposed instruction set architectures". In: *Work in Progress Session, HPCA*. January. 2000.

[8] Mark Jerome Hampton. "Exposing Datapath Elements to Reduce Microprocessor Energy Consumption". PhD thesis. Massachusetts Institute of Technology, 2001.

[9] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. "Compilation techniques for low energy: an overview". In: *Proceedings of 1994 IEEE Symposium on Low Power Electronics*. IEEE, 1994, pp. 38–39.

[10] Anil 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* (2001), p. 195.

[11] Mostafa E. A. Ibrahim, Markus Rupp, and Hossam A. H. Fahmy. "Code transformations and SIMD impact on embedded software energy/power consumption". In: *2009 International Conference on Computer Engineering & Systems* (Dec. 2009), pp. 27–32.

[12] A. Parikh et al. "Instruction scheduling based on energy and performance constraints". In: *Proceedings. IEEE Computer Society Workshop on VLSI*. IEEE Comput. Soc, 2000, pp. 37–42.

[13] L. N. Chakrapani et al. "The emerging power crisis in embedded processors: what can a poor compiler do?" In: *CASES '01 Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems* (2001).

[14] Yun Cao and Hiroto Yasuura. "A system-level energy minimization approach using datapath width optimization". In: *Proceedings of the 2001 international symposium on Low power electronics and design - ISLPED '01*. New York, New York, USA: ACM Press, 2001, pp. 231–236.

[15] R. Soma and M. Pedram. "Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times". In: *Proceedings Design, Automation and Test in Europe Conference and Exhibition* (2005), pp. 4–9.

[16] S. V. Gheorghita, Henk Corporaal, and Twan Basten. "Using iterative compilation to reduce energy consumption". In: *Proceedings of the 10th Annual Conference of the Advanced School for Computing and Imaging* (2004).

[17] Zhelong Pan and Rudolf Eigenmann. "Fast and effective orchestration of compiler optimizations for automatic performance tuning". In: *International Symposium on Code Generation and Optimization*. ii (2006), pp. 319–332.

[18] Kingsum Chow and Youfeng Wu. "Feedback-directed selection and characterization of compiler optimizations". In: *Proceedings of the Second Workshop on Feedback-Directed Optimization*. 1999, pp. 1–10.

[19] George E. P. Box, William G. Hunter, and J. Stuart Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. John Wiley & Sons, 1978, pp. 374–418.

[20] Tomasz Patyk et al. "Energy consumption reduction by automatic selection of compiler options". In: *2009 International Symposium on Signals, Circuits and Systems*. IEEE, July 2009, pp. 1–4.

[21] Grigori Fursin et al. "Milepost GCC: machine learning enabled self-tuning compiler". In: *International Journal of Parallel Programming* (2011), pp. 1–31.

[22] John Cavazos et al. "Rapidly Selecting Good Compiler Optimizations using Performance Counters". In: *International Symposium on Code Generation and Optimization (CGO'07)*. Ieee, Mar. 2007, pp. 185–197.

[23] Prasad A. Kulkarni et al. "Evaluating heuristic optimization phase order search algorithms". In: *International Symposium on Code Generation and Optimization*. 2007.

[24] S. Kulkarni and John 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.

[25] Kenneth O Stanley. "Efficient Reinforcement Learning through Evolving Neural Network Topologies". In: *Genetic and Evolutionary Computation Conference*. 2002.

[26] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. "Optimizing general purpose compiler optimization". In: *Proceedings of the 2nd conference on Computing frontiers - CF '05*. New York, New York, USA: ACM Press, 2005, p. 180.

[27] Suresh Purini and Lakshya Jain. "Finding good optimization sequences covering program space". In: *ACM Transactions on Architecture and Code Optimization* 9.4 (Jan. 2013), pp. 1–23.

[28] J. S. Seng and D. M. Tullsen. "The effect of compiler optimizations on Pentium 4 power consumption". In: *Seventh Workshop on Interaction Between Compilers and Computer Architectures, 2003. INTERACT-7 2003. Proceedings.* (2003), pp. 51–56.

[29] Mostafa E. A. Ibrahim, Markus 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.

[30] Madhavi 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).

[31] David Brooks, Vivek Tiwari, and Margaret Martonosi. "Wattch: a framework for architectural-level power analysis and optimizations". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture* (2000).

[32] J. Ayala and M. López-Vallejo. "Improving register file banking with a power-aware unroller". In: *Proceedings of PARC* (2004).

[33] YongKang Zhu et al. "The energy impact of aggressive loop fusion". In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. 2004.

[34] B. Kim, Y. Cho, and J. Hong. "An Efficient Function Inlining Scheme for Resource-Constrained Embedded Systems". In: *Journal of Information Science and Engineering* 874 (2012), pp. 859–874.

[35] M. Toburen, T. Conte, and Matt Reilly. "Instruction scheduling for low power dissipation in high performance microprocessors". In: *Proceedings of the 1998 Power Driven ...* (1998).

[36] Henry Massalin. "Superoptimizer - A Look at the Smallest Program". In: *ACM SIGARCH Computer Architecture News* (1987), pp. 122–126.

[37] Rajeev Joshi, Greg Nelson, and Keith Randall. "Denali : a goal-directed superoptimizer". In: *Proceedings of the ACM 2000 Conference on Programming Language Design and Implementation*. July. 2001, pp. 304–314.

[38] Sumit Gulwani et al. "Synthesis of loop-free programs". In: *ACM SIGPLAN Notices* 47.6 (Aug. 2012), p. 62.

[39] Sorav Bansal and Alex Aiken. "Automatic generation of peephole superoptimizers". In: *ACM SIGOPS Operating Systems Review* 40.5 (Oct. 2006), p. 394.

[40] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic superoptimization". In: *Architectural Support for Programming Languages and Operating Systems*. New York, New York, USA: ACM Press, 2013, p. 305.

[41] Vivek Tiwari et al. "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.

[42] Stefan Steinke et al. "An accurate and fine grain instruction-level energy model supporting software optimizations". In: *Proc. of PATMOS* (2001).

[43] Yongxin Zhu, Weng-Fai Wong, and tefan Andrei. "An integrated performance and power model for superscalar processor designs". In: *Proceedings of the 2005 conference on Asia South Pacific design automation - ASP-DAC '05* (2005), p. 948.

[44] N. Vijaykrishnan et al. "Energy-driven integrated hardware-software optimizations using SimplePower". In: *Proceedings of the 27th annual international symposium on Computer architecture - ISCA '00*. c. New York, New York, USA: ACM Press, 2000, pp. 95–106.

[45] G. Qu et al. "Function-level power estimation methodology for microprocessors". In: *Proceedings of the 37th ...* (2000), pp. 810–813.

[46] H Blume et al. "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.

[47] Joseph Yiu. *The Definitive Guide to the ARM Cortex-M3*. 2nd. Newnes, 2010.

[48] *The LLVM Compiler Infrastructure*. URL: http://llvm.org/.

[49] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. "Automatic selection of compiler options using non-parametric inferential statistics". In: *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)* (2005), pp. 123–132.

[50] Free Software Foundation. *The R Project for Statistical Computing*. 2013. URL: http://www.r-project.org/.

[51] U. Groemping and M. U. Groemping. *FrF2: Fractional Factorial designs with 2-level factors*. 2012. URL: ftp://www.postfix.org/mirror/postfix/samag.200001/root/mirror/CRAN/web/packages/FrF2/FrF2.pdf.

[52] Richard A. Brualdi. *Introductory Combinatorics*. 1st ed. Elvesier North-Holland, Inc., 1977, pp. 119–120.

[53] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms*. 1st ed. Addison-Wesley, 2011, pp. 416–417.

[54] Steve Kerrison and Kerstin Eder. *Energy modelling and optimisation of software for a hardware multi-threaded embedded microprocessor*. Tech. rep. Bristol: University of Bristol, 2013.

[55] Tiantian Liu et al. "Register allocation for simultaneous reduction of energy and peak temperature on registers". In: *Design, Automation & Test in Europe*. 2011.

[56] 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.

[57] Jason E. Fritts et al. "MediaBench II video: Expediting the next generation of video systems research". In: *Microprocessors and Microsystems* 33.4 (June 2009), pp. 301–318.

[58] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. "The LINPACK Benchmark: past, present and future". In: *Concurrency and Computation: Practice and Experience* 15.9 (Aug. 2003), pp. 803–820.

[59] R. P. Weicker. "Dhrystone benchmark: rationale for version 2 and measurement rules". In: *ACM SIGPLAN Notices* 23.8 (1988).

A

PhD plan

Date		Activity
Jan - Mar	2014	Continue with experiments exploring memory alignment and the effect of flash on energy consumption. Write up and submit to CASES.
Mar - Apr	2014	In collaboration with Embecosm (MAGEEC project), evaluate the energy efficient optimisation proposed, in combination with other optimisations and its effect on multiple platforms. Write up results.
Apr - Jun	2014	Explore the trade-off between flash and RAM for code execution. Explore the link with previous work done on scratchpad memory and the possibility of moving code into RAM.
Jun - Aug	2014	Using vector units with non-vectorised code is potentially more energy efficient than the non vectorised functional units. Explore this and possibly implement a compiler optimisation to utilise this.
Aug - Nov	2014	Further develop the superoptimizer. Create the optimisations database and see if any interesting energy efficient instruction sequences are created.
Nov - Mar	2015	Explore how previously developed optimisations affect each other when used simultaneously in the compiler. Explore how the compiler can intelligently choose which optimisations to apply.
Mar - Dec	2015	Write thesis
1st Jan	2016	Submit thesis

B

Activities

B.1 Presentations given and workshops attended

Name	Description
EACO workshop 2012 October	The results for the compiler optimisations analysis was presented at the EACO workshop.
NMI Every joule counts 2012 November	The results of the compiler optimisations analysis was presented at NMI EJC.
OSHUG 2012 November	A talk focused on the hardware and software frameworks behind measuring energy consumption was presented to the Open Source Hardware User Group.
LPGPU, PEGPUM 2013 January	The work on compiler optimisations was presented at the low power GPU workshop.
ENTRA meeting 2013 May	I gave a talk about superoptimisation at the ENTRA plenary meeting, to all partners.
GCC Cauldron, California 2013 July	A talk presenting the results of which compiler optimisations worked for energy consumption was presented back to the GCC compiler community.
Preparing for Parallella 2013 July	I gave a talk on parallel programming and the Epiphany chip.
OSHcamp/Wutheringbytes 2013 August	A talk was given on energy consumption and how to measure it.
ORCONF 2013 October	Attended a two-day meeting on the OpenRISC processor.
Tallinn 2013 October	Attended a HiPEAC networking event for Computing Systems week.
NMI Award Dinner 2013 November	Nominated for Young Engineer of the Year and invited by ARM to the gala dinner.

B.2 Lab demonstrating

2012/2013

- CAD Group project
- Embedded systems integration
- Introduction to computer architecture

2013/2014

- Mathematical methods for computer scientists
- Embedded systems integration
- Introduction to computer architecture

B.3 Projects

MAGEEC The MAchine Guided Energy Efficient Compilation (MAGEEC) project resulted from my work on compiler optimisations and I am heavily involved in a number of work packages for the project. I have set up the measurement framework needed to support the machine learning side of this project and work closely with the compiler engineers in Embecosm to develop new optimisations targeting energy consumption.

ENTRA The ENergy TRAnsparency project aims to expose energy consumption to the programmer without the programmer having to run the code. This is achieved by combining energy models with static analysis of the program. I have been involved with this project, suggesting benchmarks and optimisations methods. In future I will be more involved with the optimisation part of the project.