Identifying Compiler Options to Minimise Energy Consumption for Embedded Platforms

JAMES PALLISTER[1], SIMON HOLLIS[1] AND JEREMY BENNETT[2]

[1]*Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, United Kingdom.*
[2]*Embecosm, Palamos House #104, 66/67 High Street, Lymington, SO41 9AL, United Kingdom.*
Email: james.pallister@bristol.ac.uk

This paper presents an analysis of the energy consumption of an extensive number of the optimisations a modern compiler can perform. Using GCC as a test case, we evaluate a set of ten carefully selected benchmarks for five different embedded platforms.

A fractional factorial design is used to systematically explore the large optimisation space (2^{82} possible combinations), whilst still accurately determining the effects of optimisations and optimisation combinations. Hardware power measurements on each platform are taken to ensure all architectural effects on the energy consumption are captured.

We show that fractional factorial design can find more optimal combinations than relying on built in compiler settings. We explore the relationship between runtime and energy consumption, and identify scenarios where they are and are not correlated.

A further conclusion of this study is the structure of the benchmark has a larger effect than the hardware architecture on whether the optimisation will be effective, and that no single optimisation is universally beneficial for execution time or energy consumption.

1. INTRODUCTION

Energy consumption is rapidly becoming one of the most important design constraints when writing software for embedded platforms. In the hardware space there are many features, such as clock gating and dynamic frequency and voltage scaling, to reduce the power consumption of electronic devices. However, inefficient software can negate any gains from the hardware, so the combination of software and hardware must be considered together when exploring energy usage. This study focuses on processors for embedded platforms, because energy efficiency is particularly important for many of their target applications.

Optimising the software for low energy consumption is particularly important when adhering to a strict power budget. This is the case in many deeply embedded systems. In these devices the processor is a significant consumer of energy — a previous study characterised the CPU power usage of a handheld device to be between 20 and 40% of the total system power [1]. A further study, based on 45 nm technology

data from [2], calculated the power dissipation of the processors in a 64-core network on chip to be 40% of the entire system [3]. This category was the largest, ahead of memory, network, and I/O.

Compiler optimisations have the potential for energy savings with no changes to existing hardware or software — just tweaking the compiler's parameters can have a large effect on the energy consumption [4]. Sometimes this manifests in spikes of higher power followed by longer periods of lower power; at other times the power is maintained at a lower level. Both can reduce total energy. This relationship is complex, with the program, processor architecture and specific compiler options interacting. Furthermore, different optimisation passes interact with each other, so an option's efficacy cannot be tested in isolation. For example, inlining a function may mean that more effective common subexpression elimination can be performed, increasing the performance more than either option individually. Many approaches have attempted to solve this optimisation selection problem, using

Processor	Board name	RAM	Core clock	Other
ARM Cortex-M0	STM32F0DISCOVERY	8KB	48 MHz	64KB Flash
ARM Cortex-M3	STM32VLDISCOVERY	8KB	24 MHz	128KB Flash
ARM Cortex-A8	BeagleBone	256MB	500 MHz	VFP/NEON, superscalar
Adapteva Epiphany	EMEK3	32KB/core	400 MHz	FPU,superscalar,16 core NoC
XMOS L1	XK1	64KB	100 MHz	4×100MHz hardware threads

TABLE 1. The platforms explored in this paper along with some relevant details.

FIGURE 1. The hardware and software setup used to take the measurements.

techniques such as statistical methods [5], genetic algorithms [6] and iterative compilation [7]. All of these studies conclude that performance can be increased by choosing the correct set of optimisations, but exploring the space to find this set is challenging.

The energy an application takes can be measured using a set-up as shown in Fig. 1. A shunt resistor is inserted between the power supply and processor, allowing the voltage drop across it to be measured and amplified. This can be converted into an instantaneous power reading by considering the resistance of the shunt. The power logger assigns a timestamp to each power sample, allowing them to be integrated into a total amount of energy consumed during the execution of the application.

The following section covers the overall aims and hypotheses we wish to address in this paper. Then, the related work is discussed. Following this section, our approach to the problem of benchmark selection and compiler flags is given. The initial high-level results are presented with discussion of the first two hypotheses in Sect. 5. In Sect. 7, there is a short introduction to fractional factorial design, followed by the results obtained using this technique. The case studies of the most effective optimisations, and the interactions between optimisations is given in Sect. 8. Finally, concluding remarks to the application developer and the compiler writer are made.

2. OVERVIEW OF THIS WORK

The overall aim of this work is to identify compiler optimisations that are effective at reducing a benchmark's energy consumption. This is accomplished by using fractional factorial design to account for interactions between the optimisations, without having to enumerate all combinations of optimisations. This analysis is performed for multiple benchmarks and platforms, allowing general conclusions to be drawn about how the optimisations affect energy consumption.

We investigate the following hypotheses:

1. The time and energy required for a computation are always proportional to one another. We find investigate and present examples where energy and time are not correlated and explain why (Sect. 5).
2. There exists a set of compiler optimisations that gives a lower energy consumption than the predefined optimisation levels (Sect. 6).
3. It is possible to search the compiler optimisation space in an efficient and systematic manner, to assign each optimisation an overall effectiveness (Sect. 7).
4. There is no universally good optimisation across multiple benchmarks and platforms (Sect. 8).

We will evaluate the validity of these hypotheses by performing a series of practical experiments which target the set of optimisations enabled at various optimisation levels of a real compiler. This allows the optimisations to be measured for their effect on energy. Three types of experiment are performed in this study:

High-level. Each optimisation level (predefined set of optimisations) is tested for each benchmark and platform. This is a small number of tests, exploring the overall effect of the optimisation levels.

Fractional factorial design. A fractional factorial design is used to find the effectiveness of each optimisation flag defined at the optimisation level. This experiment is repeated for each optimisation level, and for each benchmark and platform combination. A large number of tests are performed for each combination of benchmark, platform and optimisation level. This is the first time this technique has been applied across multiple platforms for the purpose of analysing compiler optimisations.

Case study. Two case studies are performed. The first looks at the most effective optimisation flags across benchmarks and platforms, extracted from the fractional factorial design. The second explores the interactions between optimisations by exhaustively enumerating every combination of a small set of optimisations.

All the energy measurements in this paper are taken using physical measurement circuitry attached to the processors. This avoids the use of models which could be inaccurate, or modelling synthetic processors with no real world counterpart. A diagram of the software and hardware setup is shown in Fig. 1. By using commonly available platforms and processors, along with some more novel architectures, the results are applicable in general while still providing insight into how different types of architectures perform. It is important to consider a wide range of platforms when analysing energy consumption, since the structure of the processor's pipeline has a large effect on both the conditions in which energy consumption is high, and the types of optimisations which are effective. The platforms examined, shown in Tab. 1, have a wide range of pipeline depths, memory bandwidth capabilities and numbers of registers, allowing analysis of how an optimisation's effect on energy changes under these circumstances (an analysis of how the processor's features affect the energy consumption is given in Sect. 5).

This work makes the following contributions:

- The use of fractional factorial design to analyse a previously intractable optimisation space of GCC 4.7'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.

3.   RELATED WORK

3.1.   Compilers & Energy

To date there has been very little work that extensively explores the effect that different compiler optimisations have on energy consumption. However, there have been many studies that look at the effect of optimisations on execution time [5, 8], and several studies suggesting that execution time can be used as a proxy for energy usage [9, 10].

The topic of performance and energy being highly correlated is addressed in [11]. This work explored several different overall optimisation levels, as well as four specific optimisations, using the *Wattch simulator* [12] 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 [13], loop fusion [14], function inlining [15] and instruction scheduling [16], have been examined extensively for different platforms using both simulators and hardware measurements.

A drawback of those studies that explore energy consumption is that many of them choose to use simulators as opposed to taking hardware measurements. 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 [17] 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 consumption of the processor, including complex instruction level models [18], function-level models [19] and hybrids of these [20]. However, these all suffer the drawback that some energy consumption effects may not be modelled, potentially skewing the results.

3.2.   Optimisations Targeting 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 [21]. 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 [22] explored the possibility of using the compiler to insert `idle` instructions automatically, increasing the execution time up to a set limit. Using the SIMD pipeline has been shown to decrease energy consumption [23] by roughly 25%. Scheduling instructions to minimise the inter-instruction energy cost was evaluated to be another effective method to reduce a program's energy consumption [24]. Exploiting differences in energy consumption between other function units has been suggested in [25], where it is noted that strength reduction may use a more efficient shifter rather than a power hungry multiplier.

The use of scratchpad memory has been used to increase the energy efficiency of processors with these features in [26]. 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 [27].

3.3. Optimisation Choice

The challenge of choosing the optimisations and their order has been explored and many methodologies proposed for choosing an optimal set of optimisations.

Chakrapani et al. attempt to classify optimisations by the effect they have on performance and energy [25]. This work used both hardware measurements and a gate-level simulation to derive the results, separating the optimisations into the following three classes:

- **Reduction in energy consumption due to increase in performance.** Optimisations in this class reduce the number of cycles or instructions needed to complete the application and thus less overall work is done.
- **Optimisations that reduce energy without improving the performance.** These optimisations reduce the instantaneous power in portions of the program without increasing the number of cycles to compute the result. Scheduling instructions to reduce switching and making use of power efficient functional units often fall into this category.
- **Optimisations that increase energy consumption or decrease performance.** These include optimisations which can sometimes have unexpected performance hits, such as loop unrolling and function inlining. The increase in energy consumption can come from either a longer running time at the same average power, or a higher average power.

Iterative compilation has been examined as a possibility for choosing optimisations that reduce power by Gheorghita et al [28]. In this paper, the effect of different loop unrolling and loop tiling parameters on energy consumption is examined for three linear-algebra-based benchmarks using a simulator. The paper concluded that iterative compilation was an effective method of decreasing energy consumption as well as improving performance.

Other approaches have looked at genetic algorithms for optimisation selection [6] and optimisation phase ordering [29]. While these techniques are shown to be effective, they have the drawback that the reasons behind an optimisation's selection is not obvious. Another study [30] has explored genetic algorithms in the context of optimising multiple objectives. This study used the technique to balance the trade-off between code size, performance and worst-case performance, and could also optimise for any single objective at the cost of the others.

These techniques do not expose the relationships between optimisations, instead opting to search though the optimisation space and making a best guess about where to look next. In this paper, we improve these shortcomings by using fractional factorial design [31] to explore the most effective optimisations and the interactions between them.

Fractional factorial design, as a method for exploring the interactions of compiler optimisations was discussed in [32]. Nine optimisations were examined, using a fractional factorial technique to isolate the interactions and choose a set of optimisations that gave better performance than just enabling all the optimisations. This concept was extended by Haneda et al. [33] to combine the results of running fractional factorial design on several benchmarks into one set of flags.

A similar study conducted by Patyk et al [34], extended this work to energy efficiency. The study explored a range of GCC's options, with an aim to reduce energy consumption by identifying significant optimisations, then excluding them from further exploration using fractional factorial design. We use this technique to analyse the optimisations rather than optimise for the energy consumption.

These methods all require testing over many different compilations, which is a significant overhead when finding an optimal set. The MILEPOST GCC [35] study implemented an alternative to this, using machine learning to guess which optimisation flags would best apply to a given program. Features are extracted from the program, which are then matched against previous known results from previous compilations. This allows a set of optimisations to be predicted from just the source code. The drawback of this approach is that a large number of programs and optimisation combinations must be used to train the database, and there is not yet knowledge about the appropriate program features for predicting energy consumption or if they exist.

The majority of the studies listed in this section only examine one platform, and it is currently unknown whether their results would apply across

Name	Source	Category
2D FIR	WCET	Branching, FP
Blowfish	MiBench	Integer
CRC32	MiBench	Branching, integer
Cubic solver	MiBench	Memory, integer
Dijkstra	MiBench	Branching, integer
FDCT	WCET	Branching, memory
Float matmult	WCET	Memory, FP
Int matmult	WCET	Memory, integer
Rjindael	MiBench	Branching, integer
SHA	MiBench	Memory, integer

TABLE 2. Benchmarks selected, and the types of instructions they execute intensively (FP short for Floating Point).

several different platforms. Furthermore, iterative compilation [7] and other adaptive techniques used can leave holes of potential combinations of optimisations unexplored (due to the huge numbers of combinations possible). This can lead to the most optimal configurations not being found.

4. APPROACH

In this paper, we present an improved technique for testing the effectiveness of large numbers of compiler optimisations and their impact on energy consumption and run-times. The technique is based on the concept of fractional factorial design (see Sect. 7).

4.1. A New Benchmark Set

To explore the impact of the optimisations, a realistic set of test input programs is required. There have been many attempts to find representative programs, but none apply to a wide range of embedded processing systems. Frequently a benchmark will require host operating system support, requiring features which are rarely available on deeply embedded platforms. Also the size of the benchmark and the dataset it uses becomes of critical importance when running on platforms with such limited memory. Because of the lack of suitable suites, we evaluated each benchmark from a large number of contemporary suites. Individual benchmarks were considered for inclusion based on their distribution of instruction types. Further details on this are given below, and described in [36].

This set of 10 benchmarks, shown in Tab. 2, covers real world and synthetic applications across different aspects of the target platform. These are selected from MiBench [37] and the Worst Case Execution Time (WCET) [38] suites. Previous work on modelling the energy consumption of processors has shown that the pipelines and functional units enabled have a significant impact on the energy consumption. To cover these points, the benchmarks were characterised according to the following coarse criteria:

- **Integer pipeline intensity.** The frequency at which integer arithmetic instructions occur.
- **Floating point intensity.** The frequency of floating point operations.
- **Memory intensity.** Whether the program requires a large amount of memory bandwidth or not.
- **Branch frequency.** How often the code branches.

Similar categories of instruction types have been used previously to give a high level overview of the type of computation an application is performing [39]. Our categories group similar instruction, such as the loads and stores in MiBench, since energy consumption is predominately related to the target functional unit, rather than the specific operation.

This set of benchmarks is chosen because they do not require a host operating system. This prevents the benchmark from being pre-empted by another process and reducing the accuracy of the results. It also makes the execution of the benchmarks deterministic. For the same reasons, the benchmarks do not perform any I/O.

The benchmarks are also chosen carefully with regards to memory requirements. The benchmarks are designed to fit into the memory footprint of a wide range of embedded systems, with or without a memory hierarchy. In the cache-based systems we explore, this often has the effect that the benchmark fits entirely into a single level of cache, reducing complexity but potentially also accuracy. This is a trade-off that we have found necessary when generating benchmarks that cover a wide range of hardware platforms, and has the benefit that the benchmarks exhibit predictable memory accesses on most platforms.

4.2. Compiler Flags

We explore the impact of compiler optimisations using the GCC toolchain on the architectures shown in Tab. 1. GCC exposes its various optimisations via a number of flags that can be passed to the compiler [40]. We explore which flags have a significant impact on energy consumption and execution time.

The experiments are performed with different benchmarks, so a complete picture of architecture, optimisation and application can be seen. Using this combination, the following points of interest can be explored:

- The relationship between time and energy for our benchmarks;
- Architectural effects on energy consumption; and
- Application effects on energy consumption.

By using the techniques we have just outlined, we can rigorously evaluate our hypotheses, answering questions about the relationship between time and energy, and optimisation choice.

FIGURE 2. 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.

5.   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 high level overview of each platform and benchmark for the different optimisation levels is given in Fig. 2. 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 microarchitecture in this processor is more complex, featuring branch speculation and a larger instruction set [41].

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 [42] 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 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

FIGURE 3. Blowfish benchmark on the Cortex-M3 platform. Individual options are enabled or disabled on top of the `O1` optimisation level.

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.

6.   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 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.

FIGURE 4. Reducing a 3-factor full factorial design to a 'half fraction' design.

7.   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 [31] (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 [32], 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

FIGURE 5. Blowfish benchmark on the Cortex-M0 platform. Individual options enabled at `O1` are listed.

FIGURE 6. FDCT benchmark on the Cortex-M3 platform. Individual options enabled at `O2` are listed.

et al. in [33], 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. 4. 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 [31], 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 [34] and [5].

All FFDs used were generated by the statistical program, R [43] (a statistical programming language), using the FrF2 library [44].

7.1. 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. 5. 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. 6 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.

7.2. Efficient SIMD Units

In this section we analyse a specific case where energy consumption and execution time are not correlated.

An interesting effect is seen in 2D FIR for the Cortex-A8. The execution time decreases more than the energy

FIGURE 7.   2D FIR benchmark on the Cortex-A8 platform. Individual options enabled at O3 are listed.

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

TABLE 3.   Micro-benchmark results for multiplications on the NEON unit, with and without inter-instruction dependencies.

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. 7). 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 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, 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 [23] and shows that by using the hardware to its full capacity, the greatest energy savings can be achieved.

8. THE UNIVERSALITY OF FLAGS

We have seen large variations based on optimisation flags, and so an interesting problem for compiler designers is how to choose an optimal set of flags across different hardware platforms and applications. This section explores which individual flags had the largest effect in our experiments, our fourth hypothesis: that a consistently good optimisation is not seen across all benchmarks and platforms. Tab. 4 lists the results for this section, with the top three optimisation flags (where that optimisation has a significant effect, as per the Mann-Whitney test) identified for each benchmark and platform combination. Each letter represents an optimisation that is labelled in the table below. We also show the number of times this flag occurs.

Only 20 out of 82 (the number of flags enabled by O1, O2 and O3) options examined appear in the table. This supports the argument that many of the options have little effect on the energy consumption, and consequently performance.

For the ARM platforms, a similar set of options appears for the same benchmarks. Common options for the same benchmarks are expected, since optimisations are triggered by the structure of the source code. However, the opposite of this is seen for the Epiphany processor — there are three optimisations that are consistently effective at reducing energy. A particularly unusual option to be consistently effective is **-fdce**: dead code elimination, removing code which is never used by the application. However, this also allows the compiler to eliminate parts of the control flow graph, removing branches and decreasing the amount of work the application performs.

The optimisation listed most frequently in the table is **-ftree-dominator-opts**. The prevalence of this flag is likely due to it enabling several simple optimisation passes, performing optimisations such as copy propagation and expression simplification. Another effective optimisation is **-fomit-frame-pointer**. This optimisation frees an additional register for general use by not using a frame pointer. This optimisation is seen frequently on the ARM platforms, however not at all on the Epiphany. This is likely due to the ARM processors suffering from greater register pressure since they only have 16 registers compared to the Epiphany's 64.

We see some interesting correlations between platforms. The CRC32 benchmark does not have much optimisation potential since it consists of simple operations in a tight loop. We indeed observe that very few optimisations have a significant effect. Only one common option (**-fmove-loop-invariants**) appears across three of the four platforms. This optimisation moves redundant calculations out of loops and appears because the CRC32 benchmark has some very tight loops with redundant calculation that can be moved outside the loop regardless of platform.

As observed, some options are seen to affect the

Benchmark	Cortex-M0			Cortex-M3			Cortex-A8			Epiphany		
	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd
2dfir	E	·	·	T	G	I	N	G	B	I	A	D
blowfish	C	J	E	J	C	G	K	C	E	D	P	I
crc32	F	·	·	F	·	·	F	G	·	·	·	·
cubic	A	H	·	A	H	·	A	·	·	A	H	R
dijkstra	H	A	C	F	H	A	F	H	A	H	A	·
fdct	J	G	D	J	G	K	M	K	J	A	I	D
float_matmult	B	E	·	B	E	G	N	L	·	D	I	A
int_matmult	B	E	C	B	L	F	L	N	M	A	I	D
rijndael	C	B	O	C	B	O	K	C	S	D	K	B
sha	C	B	E	B	C	F	B	C	M	D	B	Q

ID	Count	Flag	ID	Count	Flag
A	12	`-ftree-dominator-opts`	K	5	`-fschedule-insns`
B	12	`-ftree-loop-optimise`	L	3	`-finline-small-functions`
C	11	`-fomit-frame-pointer`	M	3	`-fschedule-insns2`
D	8	`-fdce`	N	3	`-ftree-pre`
E	7	`-fguess-branch-probability`	O	2	`-ftree-sra`
F	7	`-fmove-loop-invariants`	P	1	`-fipa-profile`
G	7	`-ftree-ter`	Q	1	`-ftree-pta`
H	7	`-ftree-fre`	R	1	`-fcombine-stack-adjustments`
I	6	`-ftree-ch`	S	1	`-fgcse`
J	5	`-ftree-forwprop`	T	1	`-fpeephole2`

TABLE 4. Table showing the most effective option for each platform-benchmark combination. Options considered were optimisations enabled by `O1`, `O2` and `O3` levels.

energy consumption across benchmarks. This is due to the optimisations being targeted to specific code patterns, which only appear in some of the benchmarks. In particular we see effective options across different ARM platforms, while these same optimisations are not effective on the Epiphany. Since each of the ARM Platforms is using a slightly different instruction set (Thumb, Thumb+Thumb2 and ARM for the Cortex-M0, Cortex-M3 and Cortex-A8 respectively), we infer that the effectiveness of these options is due to commonalities between these instruction sets — namely the number of registers.

These results show the difficulty of choosing one optimisation which is good in all cases. In many cases the instruction set and micro-architecture of the processor have a large effect on how much the energy consumption is reduced. This means that a singularly good optimisation cannot be chosen.

8.1. Optimisation Chaos

In this section, we expand on the theme of there being no universally good optimisation by investigating the effect of interactions between optimisations. We conclude that there is a chaotic relationship between the platform, benchmark, and the effectiveness of the optimisation.

Examining the correlation between optimisations and their effects is a complex issue. Due to non-linear interactions, one would expect that the prediction of effects is difficult. This is borne out by our experimental results: as seen in previous Figs. 5 and 6, less than a third of the options have a significant impact. For the other optimisations, higher order interactions cause unpredictable effects, where enabling or disabling a particular optimisation can completely change the effect of many other subsequent optimisation passes.

In Fig. 3, several unexpected effects worthy of further investigation can be seen. This graph shows individual optimisations being turned on and off, using the `O1` optimisation level as a base. The flags on the left of the `O1` section were found to decrease the energy consumption when disabled, an effect not seen in the FFD results. These flags were chosen for further exploration.

To explore this inconsistency, a small case study was performed, where all combinations of four options were explored. The energy figures for exhaustive exploration can be seen in Tab. 5, with the aim being to ascertain whether the effect of this energy reduction would compound with multiple flags. The `O1` column of this table shows the results of the options applied over the `O1` optimisation level. The `O2` column shows the same but on top of the `O2` optimisation level.

From the `O1` column, this it can be seen that there are many interactions occurring between the options, as simply turning all of these options off does not decrease the energy (in fact, it increases the consumption by 1.81%). Furthermore, when disabled individually, `-fguess-branch-probability` and `-ftree-dominator-opts` decrease the energy by 2.49% and 1.76% respectively. However, when both are

X1	X2	X3	X4	O1 (mJ)	(%)	O2 (mJ)	(%)
✓	✓	✓	✓	5780	0.00	5480	0.00
×	✓	✓	✓	5640	-2.49	5540	1.00
✓	×	✓	✓	5680	-1.76	5480	-0.05
×	×	✓	✓	5730	-0.93	5620	2.49
✓	✓	×	✓	5650	-2.28	5490	0.09
×	✓	×	✓	5720	-0.97	5580	1.75
✓	×	×	✓	5610	-2.90	5480	-0.03
×	×	×	✓	5640	-2.33	5530	0.85
✓	✓	✓	×	5760	-0.34	5460	-0.43
×	✓	✓	×	5720	-1.09	5480	0.03
✓	×	✓	×	5860	1.45	5490	0.15
×	×	✓	×	5960	3.08	5480	0.00
✓	✓	×	×	5890	1.91	5470	-0.19
×	✓	×	×	5870	1.61	5570	1.57
✓	×	×	×	5690	-1.56	5480	-0.03
×	×	×	×	5880	1.81	5510	0.41

TABLE 5. Exhaustively exploring 4 options compared to O1 and O2. (Cortex-M3 with blowfish benchmark). Legend in Tab. 6.

Key	Option
X1	`-fguess-branch-probability`
X2	`-ftree-dominator-opts`
X3	`-ftree-ch`
X4	`-fif-conversion`
Abs (mJ)	Absolute energy measurement in millijoules.
O1 (%)	Percentage relative to O1.
O2 (%)	Percentage relative to O2.
✓	Optimisation is enabled.
×	Optimisation is disabled.

TABLE 6. Legend for Tab. 5.

enabled, the energy consumption (relative to O1) is only 0.93% less, worse than each flag individually.

Different results are seen entirely in the O2 column, with options that decreased energy consumption on top of O1 have little or the opposite effect when applied on top of O2.

This unpredictability suggests that these options have many interdependencies that are difficult to predict up front. It also makes choosing an optimal set of optimisations very challenging. Therefore, one of our findings is that it is very unlikely that any accurate prediction mechanism for considering an optimisation and its effect on a target system exists: the effect will always be highly dependent on the application to be used and the platform upon which it resides.

9. WHAT DOES THIS MEAN FOR THE APPLICATION DEVELOPER?

The existing collections of optimisations at the various levels do a good job of optimising for performance, and consequently, energy. These strike a good balance between ease of use and performance. However, they will never be as effective as those generated by searching through the full optimisation space. To avoid running many tests to find a good solution, developing machine-learning compiler technologies similar to MILEPOST GCC [35] would be fitting. A reasonable set of optimisations can be predicted based on high-level features and an architecture selection, and this would greatly reduce the time spent searching as demonstrated by MILEPOST GCC. This is especially true as the effectiveness and type of optimisation was found to be heavily based on the platform and the structure of the application being compiled (Sect. 8). Predicting the optimisations in this way would reduce compile times as well as the energy and execution time of the application.

This study focused on GCC, since it is a mature compiler supporting many different platforms and optimisations. As an alternative, the LLVM compiler [42] is relatively new, with a well defined set of optimisation passes, whose order can easily be specified. This extra flexibility means there may be a better solution to find, but also that it is essentially searching for a sharper needle in a bigger haystack [45]. The benefits from having this much larger space to explore may not be worth the trade-off of the time it takes to find it.

10. WHAT DOES THIS MEAN FOR THE COMPILER WRITER?

When designing a new optimisation, a compiler writer must check whether the optimisation is effective, and under what conditions. Using fractional factorial, design a compiler writer can check whether the pass is effective when combined with an arbitrary set of other optimisations. This avoids the case of the optimisation being tested in isolation, which will result in an incorrect analysis because of the interactions between optimisations. We would recommend that, when selecting optimisations to be included in a broad optimisation level, the optimisation is evaluated in this way and only selected if it has a non-negative effect over all of the benchmarks.

All the optimisations we show in this paper are designed for either performance or code size. This means we cannot draw conclusions about the effect of dedicated compiler optimisations targeting energy such as those shown in the related work (Sect. 3.2). Although all optimisation targets may be beneficial for energy usage, dedicated energy flags would have to compete against these other optimisation metrics, meaning that even if they operate well in isolation, they may not do well when grouped. There are many opportunities for further work in this area.

11. CONCLUSION

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. 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.

ACKNOWLEDGEMENTS

APPENDIX A. HARDWARE SETUP

All the measurements were taken using the INA219 power monitoring IC [46], which provides power, current and voltage outputs.

The Cortex-M0 and Cortex-M3 boards both have a single measurement point, recording the power consumed by the whole microprocessor. For the BeagleBone there are three available measurement points: the Cortex-A8 core (including caches), on-chip peripherals (power management, bus controllers) and the external SDRAM memory IC. This allows the effect of the compiler optimisations on the memory to be recorded. Adapteva's Epiphany board has two measurement points: the core power consumption and IO power consumption, whereas the XMOS board's measurement point gathers power consumption data for the core of the processor.

The hardware measurements have several sources of error. The most apparent errors are variations in the timing: the INA219 is sampled at intervals of 1 ms and the power measurement integrated over this. Small inaccuracies occur from jitter in this interval. The ADC in the INA219 also fluctuated by ± 30 μV, however this was close to the noise floor of the measurements, so had no significant effect on the results.

REFERENCES

[1] Carroll, A. and Heiser, G. (2010) An analysis of power consumption in a smartphone. *Proc. USENIX*, Boston, MA, USA, 22–25 June, pp. 21–21. USENIX Association Berkeley, CA, USA.

[2] Lotfi-kamran, P., Grot, B., Ferdman, M., Volos, S., and Kocberber, O. (2012) Scale-Out Processors. *Int. Symp. Computer Architecture 12*, Portland, Oregon, 9–13 June, pp. 500–511. IEEE Computer Society, Washington, DC, USA.

[3] Hollis, S. J., Jackson, C., Bogdan, P., and Marculescu, R. (2012) Exploiting Emergence in On-chip Interconnects. *IEEE Transactions on Computers*, **PP(99)**, 1–14.

[4] Pan, Z. and Eigenmann, R. (2006) Fast and effective orchestration of compiler optimizations for automatic performance tuning. *Int. Symp. Code Generation and Optimization 06*, New York, USA, 26–29 March, pp. 319–332. IEEE Computer Society, Washington, DC, USA.

[5] Haneda, M., Knijnenburg, P. M. W., and Wijshoff, H. A. G. (2005) Automatic selection of compiler options using non-parametric inferential statistics. *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, St. Louis, USA, 17–21 September, pp. 123–132. IEEE Computer Society, Washington, DC, USA.

[6] Lin, S. C., Chang, C. K., and Lin, N. W. (2008) Automatic selection of GCC optimization options using a gene weighted genetic algorithm. *Proc. Computer Systems Architecture Conference*, Hsinchu, Taiwan, 4–6 August, pp. 1–8. IEEE Computer Society, Washington, DC, USA.

[7] Kisuki, T., Knijnenburg, P. M. W., O'Boyle, M. F. P., Bodin, F. and Wijshoff, H. A. G. (1999) A feasibility study in iterative compilation. *Int. Symp. High Performance Computing*, Kyoto, Japan, 26–28 May, pp. 121–132. Springer Berlin Heidelberg.

[8] Purini, S. and Jain, L. (2013) Automatic selection of compiler options using non-parametric inferential statistics. *Transactions on Architecture and Code Optimization*, **9**, 1–23. ACM, New York, USA.

[9] Seng, J. S. and Tullsen, D. M. (2003) The effect of compiler optimizations on pentium 4 power consumption. *Proc. Workshop on Interaction between Compilers and Computer Architectures*, Anaheim, CA, USA, 8 Feb, pp. 51. IEEE Computer Society, Washington, DC, USA.

[10] Ibrahim, M. E. A., Rupp, M. and Habib, S. E.-D. (2009) Compiler-based optimizations impact on embedded software power consumption *Workshop on Circuits and Systems and TAISA Conference*, Toulouse, France, 28 June – 1 July, pp. 1–4. IEEE.

[11] Valluri, M. and John, L. (2001). *Is compiling for performance == compiling for power?*

[12] Brooks, D., Tiwari, V., and Martonosi, M. (2000) Wattch: a framework for architectural-level power analysis and optimizations. *Int. Symp. Computer Architecture*, Vancouver, BC, Canada, 14 June, pp. 83–94. IEEE Computer Society, Washington, DC, USA.

[13] Ayala, J. and López-Vallejo, M. (2004) Improving register file banking with a power-aware unroller. *Proc. PARC* Pisa, Italy. 15–20.

[14] Zhu, Y., Magklis, G., and Scott, M. (2004) The energy impact of aggressive loop fusion. *Proc. Int. Conf. Parallel Architectures and Compilation Techniques.* Antibes Juan-les-Pins, France, 29 Sept – 3 Oct, pp. 153–164. IEEE Computer Society, Washington, DC, USA.

[15] Kim, B., Cho, Y., and Hong, J. (2012) An Efficient Function Inlining Scheme for Resource-Constrained Embedded Systems. *Journal of Information Science and Engineering*, **28**, 859–874.

[16] Toburen, M., Conte, T., and Reilly, M. (1998) Instruction scheduling for low power dissipation in high performance microprocessors. *Proc. Power Driven Microarchitecture Workshop*, Barcelona, Spain, 28 June, pp. 14–19.

[17] Ye, W., Vijaykrishnan, N., Kandemir, M., and Irwin, M. J. (2000) The design and use of simplepower: a cycle-accurate energy estimation tool. *Proc. Design Automation Conference*, Los Angeles, California, USA, 5–9 June, pp. 340–345. ACM.

[18] Steinke, S., Knauer, M., Wehmeyer, L., and Marwedel, P. (2001) An accurate and fine grain instruction-level energy model supporting software optimizations. *Proc. PATMOS*, Yverdon-Les-Bains, Switzerland, 26–28 Sept.

[19] Qu, G., Kawabe, N., Usami, K., and Potkonjak, M. (2000) Function-level power estimation methodology for microprocessors. *Proc. Design Automation Conference*, Los Angeles, CA, USA, 5–9 June, pp. 810–813. ACM.

[20] Blume, H., Becker, D., Rotenberg, L., Botteck, M., Brakensiek, J., and Noll, T. (2007) Hybrid functional- and instruction-level power modeling for embedded and heterogeneous processor architectures. *Journal of Systems Architecture*, **53**, 689–702.

[21] Tiwari, V., Malik, S., and Wolfe, A. (1994) Compilation techniques for low energy: an overview. *IEEE Symp. Low Power Electronics*, San Diego, CA, USA, 10–12 Oct, pp. 38 –39. IEEE Computer Society, Washington, DC, USA.

[22] Seth, A., Keskar, R. B., and Venugopal, R. (2001) Algorithms for energy optimization using processor instructions. *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, Atlanta, Georgia, USA, pp. 195–202. ACM.

[23] Ibrahim, M. E. A., Rupp, M., and Fahmy, H. A. H. (2009) Code transformations and SIMD impact on embedded software energy/power consumption. *Proc. Int. Conf. Computer Engineering & Systems*, Cairo, Egypt, 14–16 Dec, pp. 27–32. IEEE Computer Society, Washington, DC, USA.

[24] Parikh, A., Kandemir, M., Vijaykrishnan, N., and Irwin, M. (2000) Instruction scheduling based on energy and performance constraints. *Proc. IEEE Computer Society Workshop on VLSI*, Orlando, FL, USA, pp. 37–42. IEEE Computer Society, Washington, DC, USA.

[25] Chakrapani, L. N. and et al. (2001) The emerging power crisis in embedded processors: What can a (poor) compiler do? *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, Atlanta, Georgia, USA, pp. 176–180. ACM.

[26] Steinke, S., Wehmeyer, L. and Marwedel, P. (2002) Assigning program and data objects to scratchpad for energy reduction. *Proc. Design Automation and Test in Europe*, Paris, France, 4–8 March, pp. 409–415. IEEE.

[27] Cao, Y. and Yasuura, H. (2001) A system-level energy minimization approach using datapath width optimization. *Proc. Int. Symp. Low Power Electronics and Design*, California, USA, 6–7 August, pp. 231–237. IEEE.

[28] Gheorghita, S. V., Corporaal, H., and Basten, T. (2005) Iterative compilation for energy reduction. *J. Embedded Computing*, **1**, 509–520.

[29] Almagor, L., Cooper, K. D., and Grosul, A. (2004) Finding effective compilation sequences. *Proc. ACM Conf. Languages, Compilers, and Tools for Embedded Systems*, Washington, DC, USA, 11–13 June, pp. 231–239. ACM.

[30] Lokuciejewski, P., Plazar, S., Falk, H., Marwedel, P. and Thiele, L. (2011) Approximating Pareto optimal compiler optimization sequences — a trade-off between WCET, ACET and code size. *Software — Practice and Experience*, **41**, 1437–1458.

[31] George E. P. Box, J. S. H., William G. Hunter (1978) *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building.* John Wiley & Sons, New York.

[32] Chow, K. and Wu, Y. (1999) Feedback-directed selection and characterization of compiler optimizations. *Workshop on Feedback Directed Optimization*, Austin, Texas, USA, 1 Dec.

[33] Haneda, M. Knijnenburg, P. M. W. and Wijshoff, H. A. G. (2005) Optimizing General Purpose Compiler Optimization. *Proc. 2nd Conf. Computing Frontiers*, Ischia, Italy, 4–6 May, pp. 180–188. ACM.

[34] Patyk, T., Hannula, H., Kellomaki, P., and Takala, J. (2009) Energy consumption reduction by automatic selection of compiler options. *Proc. Int. Symp. Signals, Circuits and Systems*, Iasi, Romania, 9–10 July, pp. 1–4. IEEE Computer Society, Washington, DC, USA.

[35] Fursin, G., Kashnikov, Y., and Memon, A. W. (2011) Milepost GCC: machine learning enabled self-tuning compiler. *Int. J. Parallel Programming*, **39**, 296–327.

[36] Pallister, J., Hollis, S. and Bennett, J. (2013) BEEBS: Open benchmarks for energy measurements on embedded platforms. [Preprint] Available from: http://www.cs.bris.ac.uk/Research/Micro/beebs.jsp [Accessed 23rd August 2013].

[37] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001) Mibench: A free, commercially representative embedded benchmark suite. *Proc. IEEE Workshop on Workload Characterization*, Washington, DC, USA, pp. 3–14. IEEE Computer Society, Washington, DC, USA.

[38] Gustafsson, J., Betts, A., Ermedahl, A., and Lisper, B. (2010) The mälardalen wcet benchmarks - past, present and future. *Proc. 10th Int. Workshop on Worst-Case Execution Time Analysis*, Brussels, Belgium, 6 July, pp. 137–147.

[39] Hennessy, J. L. and Patterson, D. A. (2012) *Computer Architecture: A Quantitative Approach*, 5th edition. Morgan Kaufmann, MA, USA.

[40] Free Software Foundation, GCC, the GNU Compiler Collection, http://gnu.gcc.org/, (Accessed 2013/03/20).

[41] Yiu, J. (2010) *The Definitive Guide to the ARM Cortex-M3*, 2nd edition. Newnes, MA, USA.

[42] University of Illinois. The LLVM Compiler Infrastructure, http://llvm.org/, (Accessed 2013/03/20).

[43] Free Software Foundation, The R Project for Statistical Computing, http://www.r-project.org/, (Accessed 2013/03/20).

[44] Groemping, U. and Groemping, M. U. (2012). FrF2: Fractional Factorial designs with 2-level factors.

[45] Kulkarni, Prasad A., Whalley, David B., Tyson, Gary S. and Davidson, Jack W. (2007) Evaluating heuristic optimization phase order search algorithms. *Int. Symp. Code Generation and Optimization*, California, USA, 11–14 March, pp. 157–169. IEEE.

[46] Texas Instruments. (2011) *INA219: Current / Power Monitor with I2C^{TM} Interface*. Texas Instruments, Dallas, Texas, USA.