Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms

James Pallister1,*, Simon J. Hollis1 and Jeremy Bennett2

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

This paper presents an analysis of the energy consumption of an extensive number of the optimizations a modern compiler can perform. Using GCC as a test case, we evaluate a set of 10 carefully selected benchmarks for 5 different embedded platforms. A fractional factorial design is used to systematically explore the large optimization space (282 possible combinations), while still accurately determining the effects of optimizations and optimization 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 run-time 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 optimization will be effective, and that no single optimization is universally beneficial for execution time or energy consumption.

Keywords: compilation; energy optimization; optimization selection; fractional factorial design; energy efficiency; embedded systems; benchmarks; power measurement

Received 10 April 2013; revised 24 August 2013
Handling editor: Fionn Murtagh

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.

Optimizing 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 characterized 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 optimizations 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 optimization 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
We investigate the following hypotheses:

1. The time and energy required for a computation are always proportional to one another. We investigate and present examples where energy and time are not correlated and explain why (Section 5).

2. There exists a set of compiler optimizations that gives a lower energy consumption than the predefined optimization levels (Section 6).

3. It is possible to search the compiler optimization space in an efficient and systematic manner, to assign each optimization an overall effectiveness (Section 7).

4. There is no universally good optimization across multiple benchmarks and platforms (Section 8).

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

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

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

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

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 optimizations that are effective.
platforms examined, shown in Table 1, have a wide range of pipeline depths, memory bandwidth capabilities and numbers of registers, allowing analysis of how an optimization’s effect on energy changes under these circumstances (an analysis of how the processor’s features affect the energy consumption is given in Section 5).

This work makes the following contributions:

(i) The use of fractional factorial design to analyse a previously intractable optimization space of GCC 4.7’s optimization options.
(ii) Analysis of relative importance of each optimization across multiple benchmarks and platforms.
(iii) The answers to the previously given hypotheses.
(iv) Commentary on how these techniques and results can be used by application developers and compiler writers.

3. RELATED WORK

3.1. Compilers and energy

To date there has been very little work that extensively explores the effect that different compiler optimizations have on energy consumption. However, there have been many studies that look at the effect of optimizations 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 optimization levels, as well as four specific optimizations, using the Watch simulator [12] to estimate energy results. However, the specific optimizations were all applied individually on top of the first optimization level, without exploring any possible interactions between the optimizations. The main conclusion drawn from this study was that most optimizations reduce the number of instructions executed, hence reducing energy consumption and execution time simultaneously.

Of the studies that look at individual optimizations and their effects on energy or power, most focus on only a few optimizations in isolation and few consider multiple platforms with different architectural features. Commonly explored optimizations, 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 Watch 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, Watch 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, all these suffer the drawback that some energy consumption effects may not be modelled, potentially skewing the results.

3.2. Optimizations targeting energy

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

An analysis of the techniques the compiler can perform to optimize for energy was carried out by Tiwari et al. [21]. They identified several possible techniques that compilers could use to reduce the energy consumption of programs. They were:

(i) Reorder instructions to reduce switching.
(ii) Reduce switching on address lines.
(iii) Reduce memory accesses.
(iv) Improve cache hits.
(v) Improve page hits.
The last three will also normally increase performance as well as reduce energy.

Several novel types of compiler optimizations 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 minimize 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. Optimization choice

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

Chakrapani et al. attempt to classify optimizations 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 optimizations into the following three classes:

(i) Reduction in energy consumption due to increase in performance. Optimizations in this class reduce the number of cycles or instructions needed to complete the application and thus less overall work is done.

(ii) Optimizations that reduce energy without improving the performance. These optimizations 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.

(iii) Optimizations that increase energy consumption or decrease performance. These include optimizations 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 optimizations 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 optimization selection [6] and optimization phase ordering [29]. While these techniques are shown to be effective, they have the drawback that the reasons behind an optimization’s selection are not obvious. Another study [30] has explored genetic algorithms in the context of optimizing multiple objectives. This study used the technique to balance the trade-off between code size, performance and worst-case performance, and could also optimize for any single objective at the cost of the others.

These techniques do not expose the relationships between optimizations, instead opting to search through the optimization space and making the 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 optimizations and the interactions between them.

Fractional factorial design, as a method for exploring the interactions of compiler optimizations was discussed in [32]. Nine optimizations were examined, using a fractional factorial technique to isolate the interactions and choose a set of optimizations that gave better performance than just enabling all the optimizations. 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 optimizations, then excluding them from further exploration using fractional factorial design. We use this technique to analyse the optimizations rather than optimize for the energy consumption.

All these methods 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 optimization 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 optimizations to be predicted from just the source code. The drawback of this approach is that a large number of programs and optimization 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 several different platforms. Furthermore, iterative compilation [7] and other adaptive techniques used can leave holes of potential combinations of optimizations 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 optimizations and their impact on energy consumption and run-times. The technique is based on the concept of fractional factorial design (see Section 7).

4.1. A new benchmark set

To explore the impact of the optimizations, 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 Table 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 characterized according to the following coarse criteria:

(i) **Integer pipeline intensity.** The frequency at which integer arithmetic instructions occur.
(ii) **Floating point intensity.** The frequency of floating point operations.
(iii) **Memory intensity.** Whether the program requires a large amount of memory bandwidth or not.
(iv) **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 regard 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 system we explore (the Cortex-A8), 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 optimizations using the GCC toolchain on the architectures shown in Table 1. GCC exposes its various optimizations 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, optimization and application can be seen. Using this combination, the following points of interest can be explored:

(i) the relationship between time and energy for our benchmarks;
(ii) architectural effects on energy consumption; and
(iii) 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 optimization choice.

5. TIME AND ENERGY

The following section addresses the first hypothesis, and shows 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...
optimization levels is given in Fig. 2. This figure shows a line graph for each combination, displaying the effect of the broad optimization levels $O_0$ to $O_4$, $O_4$ is $O_3$ with link-time optimization. The last point is $O_5$—optimize for space. Some results are unavailable for when the compiler crashed while producing the output binary.

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 [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 optimizations. 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 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 $O_2$ 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 optimization impact.
6. OPTIMIZATION POTENTIAL

The second hypothesis to explore is that it was possible to find a set of optimizations that perform better than the standard optimization levels. Figure 3 shows each option in $O_1$ and $O_2$ optimization levels enabled on top of the flags in $O_1$. 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 optimizations that performs better than the predefined $O_1$ optimization level.

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

7. FRACTIONAL FACTORIAL DESIGN

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

GCC has over 150 different options that can be enabled to control optimizations. The majority of these options are binary—the optimization pass is either enabled or disabled. To further complicate matters, an optimization 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 optimizations 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 [33], where an FFD 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-hand side 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 allows the difference between any two factors to be estimated.

The drawback of 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 the fourth-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 [5, 34].

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

---

FIGURE 3. Blowfish benchmark on the Cortex-M3 platform. Individual options are enabled or disabled on top of the $O_1$ optimization level.

---
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 optimization 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.

Figure 6 highlights a discrepancy that occurred between execution time and energy consumption, even for very similar optimizations. The first two options listed (-fschedule-insns and -fschedule-insns2) both schedule instructions to
FIGURE 6. FDCT benchmark on the Cortex-M3 platform. Individual options enabled at O2 are listed.

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 recognizing 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 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 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 optimization 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 Table 3, showing that performing 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 optimization 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 optimization is not seen across all benchmarks and platforms. Table 4 lists the results for this section, with the top three optimization flags (where that optimization has a significant effect, as per the Mann–Whitney test) identified for each benchmark and
platform combination. Each letter represents an optimization 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 $O_1$, $O_2$ and $O_3$) 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 optimizations are triggered by the structure of the source code. However, the opposite of this is seen for the Epiphany processor—there are three optimizations 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 optimization listed most frequently in the table is -ftree-dominator-opts. The prevalence of this flag is likely due to it enabling several simple optimization passes, performing optimizations such as copy propagation and expression simplification. Another effective optimization is -fomit-frame-pointer. This optimization frees an additional register for general use by not using a frame pointer. This optimization 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 have only 16 registers compared to the Epiphany’s 64.

We see some interesting correlations between platforms. The CRC32 benchmark does not have much optimization potential since it consists of simple operations in a tight loop. We indeed observe that very few optimizations have a significant effect. Only one common option (-fmove-loop-invariants) appears across three of the four platforms. This optimization 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 energy consumption across benchmarks. This is due to the optimizations being targeted to specific code patterns, which only appear in some of the benchmarks. In particular, we see effective options across different code patterns, while these same optimizations 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
TABLE 4. Table showing the most effective option for each platform-benchmark combination. Options considered were optimizations enabled by O1, O2 and O3 levels.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Cortex-M0</th>
<th>Cortex-M3</th>
<th>Cortex-A8</th>
<th>Epiphany</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1st</td>
<td>2nd</td>
<td>3rd</td>
<td>1</td>
</tr>
<tr>
<td>2dfir</td>
<td>E</td>
<td>-</td>
<td>T</td>
<td>G</td>
</tr>
<tr>
<td>blowfish</td>
<td>C</td>
<td>J</td>
<td>E</td>
<td>J</td>
</tr>
<tr>
<td>crc32</td>
<td>F</td>
<td>-</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>dijkstra</td>
<td>H</td>
<td>A</td>
<td>C</td>
<td>F</td>
</tr>
<tr>
<td>float матmult</td>
<td>B</td>
<td>E</td>
<td>-</td>
<td>B</td>
</tr>
<tr>
<td>int матmult</td>
<td>B</td>
<td>E</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>rjndael</td>
<td>C</td>
<td>B</td>
<td>O</td>
<td>C</td>
</tr>
<tr>
<td>sha</td>
<td>C</td>
<td>B</td>
<td>E</td>
<td>B</td>
</tr>
</tbody>
</table>

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

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 optimization 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 optimization cannot be chosen.

8.1. Optimization chaos

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

Examining the correlation between optimizations 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 optimizations, higher-order interactions cause unpredictable effects, where enabling or disabling a particular optimization can completely change the effect of many other subsequent optimization passes.

In Fig. 3, several unexpected effects worthy of further investigation can be seen. This graph shows individual optimizations being turned on and off, using the O1 optimization 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 Table 5 (with Table 6 giving definitions of the notation used), 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 optimization level. The O2 column shows the same but on top of the O2 optimization level.

From the O1 column, 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
TABLE 5. Exhaustively exploring four options compared to 01 and 02. (Cortex-M3 with blowfish benchmark). The definition of the notation used is given in Table 6.

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

TABLE 6. Definition of notation used in Table 5.

<table>
<thead>
<tr>
<th>Key</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>-fguess-branch-probability</td>
</tr>
<tr>
<td>X2</td>
<td>-ftree-dominator-opts</td>
</tr>
<tr>
<td>X3</td>
<td>-ftree-ch</td>
</tr>
<tr>
<td>X4</td>
<td>-fif-conversion</td>
</tr>
<tr>
<td>Abs (mJ)</td>
<td>Absolute energy measurement in millijoules.</td>
</tr>
<tr>
<td>O1 (%)</td>
<td>Percentage relative to O1.</td>
</tr>
<tr>
<td>O2 (%)</td>
<td>Percentage relative to O2.</td>
</tr>
<tr>
<td>✓</td>
<td>Optimization is enabled.</td>
</tr>
<tr>
<td>×</td>
<td>Optimization is disabled.</td>
</tr>
</tbody>
</table>

Disabled individually, -fguess-branch-probability and -ftree-dominator-opts decrease the energy by 2.49% and 1.76%, respectively. However, when both are 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 optimizations very challenging. Therefore, one of our findings is that it is very unlikely that any accurate prediction mechanism for considering an optimization 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 optimizations at the various levels do a good job of optimizing 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 optimization 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 optimizations 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 optimization was found to be heavily based on the platform and the structure of the application being compiled (Section 8). Predicting the optimizations 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 optimizations. As an alternative, the LLVM compiler [42] is relatively new, with a well-defined set of optimization passes, whose order can easily be specified. This extra flexibility means that 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 additional benefits that may be gained from exploring this much larger space 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 optimization, a compiler writer must check whether the optimization is effective, and under what conditions. Using fractional factorial, design a compiler writer that can check whether the pass is effective when combined with an arbitrary set of other optimizations. This avoids the case of the optimization being tested in isolation, which will result in an incorrect analysis because of the interactions between optimizations. We would recommend that, when selecting optimizations to be included in a broad optimization level, the optimization is evaluated in this way and only selected if it has a non-negative effect over all of the benchmarks.

All the optimizations we show in this paper are designed for either performance or code size. This means that we cannot draw conclusions about the effect of dedicated compiler optimizations targeting energy such as those shown in the related work (Section 3.2). Although all optimization targets may be beneficial for energy usage, dedicated energy flags would have to compete against these other optimization 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 optimization levels. The more detailed fractional factorial design runs also demonstrated this result, showing that most optimizations had the same relative effect on energy and time. This result occurs because the majority of optimizations focus on reducing the total amount of work performed by the benchmarks—thus minimizing both energy consumption and execution time.

By adding and subtracting individual flags on top of the whole optimization 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 optimization space to gain information about the effectiveness of each optimization. To perform this we leveraged fractional factorial designs, allowing us to test each optimization 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 optimization which was effective for all benchmarks and platforms was evaluated using fractional factorial designs. We were able to extract the most effective optimizations for each benchmark and platform pair and these results showed that there was no single optimization that was universally effective. Further analysis of adding and subtracting individual flags showed that the optimization space is chaotic, with optimizations interacting in unpredictable ways.

The compiler writer can use these results and the fractional factorial design method to evaluate potential optimization passes, ensuring that they perform well in a variety of configurations. Until a method for resolving the interactions between optimizations is found, it is envisioned that the developer could use this technique to eliminate optimizations 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.

FUNDING

This study was funded by Embecosm. The original research proposal was a result of the Energy Aware Computing (EACO) workshops at the University of Bristol, sponsored by the Institute for Advanced Studies. This study was also partly sponsored by EPSRC’s Doctoral Training Account EP/K502996/1 (to J.P.). The Epiphany development kit was provided by Adapteva. Funding to pay the Open Access publication charges for this article was provided by Embecosm.

REFERENCES


APPENDIX. 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 optimizations 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 $\pm 30 \mu V$, however this was close to the noise floor of the measurements, so had no significant effect on the results.