Is Compiling for Performance == Compiling for Power?

Madhavi Valluri and Lizy John
Laboratory for Computer Architecture
Department of Electrical and Computer Engineering
The University of Texas at Austin
valluri@ece.utexas.edu
ljohn@ece.utexas.edu

Abstract

Energy consumption and power dissipation are increasingly becoming important design constraints in high performance microprocessors. Compilers traditionally are not exposed to the energy details of the processor. However, with the increasing power/energy problem, it is important to evaluate how the existing compiler optimizations influence energy consumption and power dissipation in the processor. In this paper we present a quantitative study wherein we examine the effect of the standard optimizations levels `-O1` *to* `-O4` *of DEC Alpha's* cc *compiler on power and energy of the processor. We also evaluate the effect of four individual optimizations on power/energy and attempt to classify them as "low energy" or "low power" optimizations. In our experiments we find that optimizations that improve performance by reducing the number of instructions are optimized for energy. Such optimizations reduce the total amount of work done by the program. This is in contrast to optimizations that improve performance by increasing the overlap in the program during execution. The latter kind of optimizations increase the average power dissipated in the processor.*

1 Introduction

Energy consumption and power dissipation are increasingly becoming important design constraints in high performance microprocessors. Power dissipation affects circuit reliability and packaging costs. Energy consumption directly effects battery life. With the increasing use of general purpose processors in the embedded world, designing low energy processors is important. Gowan et al. [5], discuss the power and energy trends of three generations of Alpha processors. Power dissipation increases significantly from one generation to the next despite the reduced supply voltages and advanced processor technologies. The paper shows the power in the Alpha 21264 increasing almost linearly with frequency, with power reaching 72 Watts at 600MHz. The maximum power dissipated under worst case conditions was found to be about 95 Watts. These examples clearly indicate that power dissipation and energy consumption will soon become important limiting factors in the design of high performance processors.

Until recently, the two problems were being dealt with only at the circuit-level. Voltage scaling, low swing buses, conditional clocking etc have helped alleviate the problems enormously. However, architectural-level and compiler-level analysis can help tackle these problems much earlier in the design cycle. Recently, several architectural and compiler techniques have been proposed to reduce power and energy [3, 6, 7, 8, 9, 10, 11, 12]. In our work we concentrate on the influence of compilers on power dissipation and energy consumption.

Compilers traditionally are not exposed to the energy details of the processor. Current compiler optimizations are tuned primarily for performance and occasionally for code size. With the increasing power/energy problem, it is important to evaluate how the existing optimizations influence energy consumption and power dissipation in the processor. An interesting question to answer would be - if we compile for performance, are we automatically compiling for low power or low energy? Current compilers already have two axes in the optimizations used - namely compiling for speed(in general-purpose processors) and compiling for code size(in embedded systems), do we need a third axis with optimizations that *compile for power/energy*?

To answer the above questions, we present a quantitative study wherein we examine the influence of a few state-of-the-art compiler optimizations on energy and power of the complete processor. We study the effect of the standard optimizations levels `-O1` to `-O4` of DEC Alpha's *cc* compiler on power and energy of the processor. We also evaluate the effect of four individual optimizations on power/energy and attempt to classify them as "low energy optimizations" or "low power optimizations" or both. The optimizations we study are *simple basic-block scheduling, loop unrolling, function inlining, and aggressive global scheduling*. For our experiments, we use Wattch [2], an architectural simulator that estimates CPU energy consumption. Wattch integrates parameterizable power models into the *Simplescalar* [4] processor simulator.

In our study we find that the set of compiler optimizations that improve performance by reducing the number of instructions executed are optimized for both energy and power. This is in contrast to optimizations that improve performance by increasing the existing parallelism in the program. The latter kind of optimizations increase the average power dissipated in the processor. We find that optimizations such as common-subexpression elimination, copy propagation, loop unrolling are very good for reducing energy since they reduce the number of instructions in the program, hence the amount of total work done is less in programs with these optimizations. Such optimizations should definitely be included in the *compile for power/energy* switch. Optimizations such as instruction scheduling significantly increase power (and may occasionally increase energy) because they increase the overlap in programs without reducing the total number of instructions in the program. However, such optimizations can be easily modified to take power details into consideration and can be used to increase performance without increasing average power.

The rest of the paper is organized as follows: In Section 2, we discuss some previous work that has been done in the area of compilers and low power/energy. Section 3 shows a few examples that motivates the need for our study. We describe the different compiler optimizations evaluated in Section 4. In Section 5, we describe our experimental framework and discuss in detail the results obtained. Finally, we provide concluding remarks and future directions in Section 6.

2 Related Work

In this section we present some of the previous work done in understanding the interaction between the compiler and power/energy of the processor.

The study by Kandemir et al. [7] quantitatively examines the influence of different high-level compiler optimizations on system energy. However, in their study, they evaluate only loop-nest optimizations such as loop fusion, loop fission, blocking, tiling, scalar expansion and unrolling. In our paper, we discuss both the power dissipated and energy consumption details, while in the paper by [7] they report only energy details. Their main observation in the paper is that the optimizations appear to increase the energy consumed in the core while reducing the energy consumed in the memory system. Unoptimized codes consume more energy in the memory system.

There have been a few instruction scheduling techniques proposed which attempt to reduce the power dissipated in the processor. Su et al. [11] proposed *cold scheduling*, wherein, they assign priority to instructions based on some pre-determined power cost and use a generic list scheduler to schedule the instructions. The power cost of scheduling an instruction depends on the instruction it is being scheduled after. This corresponds to the switching activity on the control path. Toburen et al. [12] propose another power-aware scheduler which schedules as many instructions as possible in a given cycle until the energy threshold of that cycle is reached. Once that precomputed threshold is reached, scheduling proceeds to the next time-step or cycle. In our work, by evaluating several state-of-the-art optimizations, we attempt to identify other optimizations besides instruction scheduling that can be improved if the power/energy models of the processor were exposed to them.

Significant work has been done in reducing energy consumption in the memory. Most techniques achieve a reduction in energy through innovative architectural techniques [6, 8, 9, 10]. Some of the works that include compiler involvement are [10] and [6]. In [6], the authors suggest the use of an L-cache. An L-cache is a small cache which is placed between the I-cache and CPU. The L-cache is very small (holds a few basic blocks), hence consumes less energy. The compiler is used to select good basic blocks to place in the L-cache. Another approach to reduce memory energy is Gray code addressing [10]. This form of addressing reduces the bit switching activity in the instruction address path. Bunda et al. [3] and Asanovic [1] investigated the effect of energy-aware instruction sets. These techniques would involve the compiler even earlier in the code generation process. The paper by Bunda et al [3] concentrates on reducing memory energy, and Asanovic [1] investigates new instructions to reduce energy in the memory, register files and pipeline stages.

3 Motivating Examples

Consider the data dependence graph (DDG) shown in Figure 1(b). It contains six operations. All operations except `op E` have a latency of one cycle, `op E` takes two cycles to complete. We will assume there are infinite functional units for this example. An instruction scheduler that attempts to also optimize for registers would schedule `op E` as close to `op F` as possible. The resulting schedule can be seen in Figure 1(b). If we assume that each operation consumes one unit of power, compared to the schedule in Figure 1(b), the schedule in Figure 1(c), dissipates less peak power (3 units vs 2 units in Figure 1(b)). Figure 1(c) is also a valid schedule. By extending the lifetime of `op E` by one cycle, we reduce the peak power dissipated without affecting performance. The design choice of letting `op E` occupy the register for one cycle longer than required will prove to be inexpensive only if there are sufficient number of registers. Current schedulers do not take power details into consideration and hence might schedule `op E` in cycle 2 even if there are sufficient registers. This example shows that two variations of the same code can have the same performance but different power requirements.

Another good candidate for reducing energy without increasing power would be function-in-lining. Function-in-lining is done in cases where the callee procedure body is small. In these cases, the code required for the calling sequences outweigh the code in the procedure body. If this procedure is called many times, in-lining can save a tremendous number of instructions. Function-in-lining does not increase the overlap such as instruction scheduling, hence this optimization keeps energy low and holds the power constant. This optimization can be a good candidate to use in the "compile for power/energy" switch.

These examples show that compilers can be optimized to produce code for low power or low energy, without sacrificing performance. In this study we hope to expose the current void in the area of power/energy-aware compilers and attempt to identify good candidates for further improvement.

4 Compiler Optimizations

In our study we evaluate the influence of compiler optimizations on processor power/energy using the native C compiler *cc* on a Dec Alpha 21064 running the OSF1 operating system. We also used the *gcc* compiler to study the effect of a few individual optimizations. The details of both the compilers and their different options are presented in the following subsections.

4.1 Standard Optimization Levels on *cc* and *gcc*

The different levels in the *cc* compiler, along with the optimizations performed at each level are described below.

`-O0` No optimizations performed. In this level, the compiler's goal is to reduce the cost of compilation. Only variables declared register are allocated in registers.

`-O1` Many local optimizations and global optimizations are performed. These include recognition and elimination of common subexpressions, copy propagation, induction variable elimination, code motion, test replacement, split lifetime analysis, and some minimal code scheduling.

`-O2` This level does inline expansion of static procedures. Additional global optimizations that improve speed (at the cost of extra code size), such as integer multiplication and division expansion (using shifts), loop unrolling, and code replication to eliminate branches are also performed. Loop unrolling and elimination of branch instructions increase the size of the basic blocks. This helps the hardware exploit instruction level parallelism (ILP) in the program.

`-O3` Includes all `-O2` optimizations and also does inline expansion of global procedures performed.

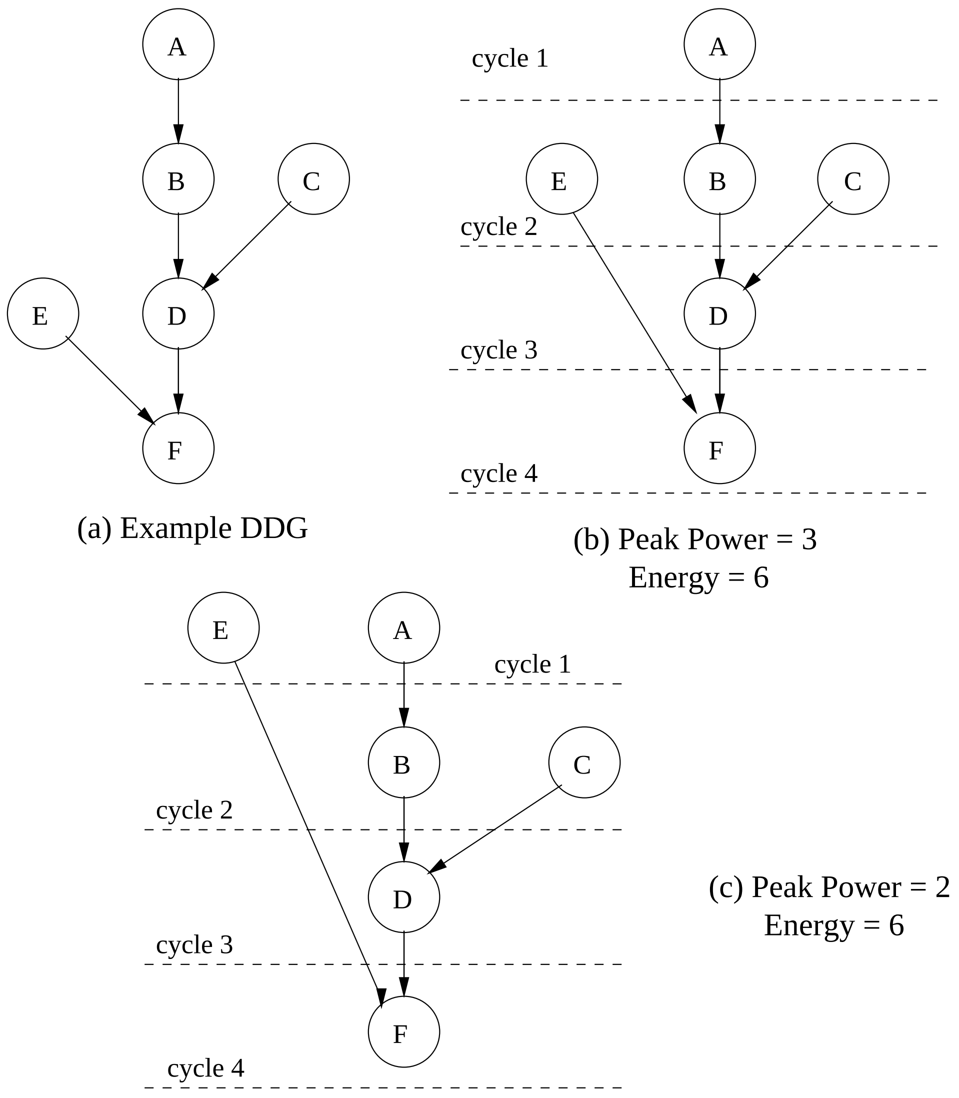

Figure 1: Motivating Example

`-O4` Software pipelining, an aggressive instruction scheduling technique used to exploit ILP in loops is performed using dependency analysis. Vectorization of some loops on 8-bit and 16-bit data is also done. This level also invokes a scheduling pass which inserts NOP instructions to improve the scheduling.

We use the FORTRAN *g77* compiler to compile the SpecFP benchmarks. *g77* is a program to call *gcc* with options to recognize programs written in Fortran. The standard optimization levels offered by *gcc* are listed below:

`-O0` No optimizations performed.

`-O1` This level is very similar to the `-O1` in *cc*. Optimizations performed are common subexpression elimination, combining instruction through substitution (copy propagation), dead-store elimination, loop strength reduction and minimal scheduling.

`-O2` Nearly all supported optimizations that do not involve a space-speed tradeoff are performed. Loop unrolling and function inlining are not done, for example. This level also includes an aggressive instruction scheduling pass.

`-O3` This turns on everything that `-O2` does, along with also inlining of procedures.

We note that in both *cc* and *gcc*, the optimizations that increase the ILP in a program are in optimization levels `-O2, -O3 and -O4` (`-O4` only in *cc*). The different levels include almost the same optimizations in both the compilers. We use both *cc* and *gcc* in our work. We use *cc* wherever possible, *gcc* wherever specific hooks to control individual optimizations are required.

4.2 Individual Optimizations

We analyze the impact of four different individual optimizations provided by *gcc*. We chose *gcc* for this because *gcc* provides more number of distinct individual optimizations than *cc* to chose from. All the individual optimizations are applied on top of optimizations performed at `-O1`. The individual optimizations chosen are:

-fschedule-insns This optimization attempts to reorder instructions to eliminate execution stalls that occur due unavailability of required data. This helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating point instruction is required. The scheduler used is a basic-block list-scheduler and it is run after local register allocation has been performed.

-fschedule-insns2 Similar to *-fschedule-insns*, but requests an additional pass of instruction scheduling after register allocation has been done. This pass does aggressive global scheduling before and after global register allocation. Postpass scheduling (when scheduling is done after register allocation) minimizes the pipeline stalls due to the spill instructions introduced by register allocation.

-finline-functions Integrates all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

-funroll-loops Perform the optimization of loop unrolling. This is done only for loops whose number of iterations can be determined at compile time or run time.

5 Experimental Results

In this section we first describe the Wattch simulator and our benchmarks. We then present a detailed analysis of our results.

5.1 Wattch 1.0 and Benchmarks

We use the Wattch 1.0 simulator [2] for our experimentation. Wattch is an architectural simulator that estimates CPU energy consumption. The power/energy estimates are based on a suite of parameterizable power models for various hardware structures in the processor and on the resource usage counts. The power models are interfaced with *Simplescalar* [4]. `sim-outorder`, *Simplescalar's* out-of-order issue simulator has been modified to keep track of which unit is being accessed in each cycle and record the total energy consumed for an application.

Wattch has three different options for clock gating to disable unused resources in the processor. The simplest clocking style assumes that the full modeled power will be consumed if any accesses occur in a given cycle, and zero otherwise. This is ideal clock gating. The second possibility assumes that if only a portion of a unit's port are accessed, the power is scaled linearly according to the number of ports being used. In the third clock gating scheme, power is scaled linearly with port or unit usage, but unused units dissipate 10% of their maximum power. This corresponds to the static power dissipated when there is no activity in unit. We chose power and energy results corresponding to the third scheme since it is the most realistic of all schemes. We used the default configuration in `sim-outorder` for our study, but changed the RUU (Register Update Unit) from 16 to 32 and LSQ (Load Store Queue) LSQ size from 8 to 16. The functional unit latencies exactly match the functional units latencies in the Alpha 21064 processor. We use the process parameters for a .35um process at 600MHz.

We chose six different benchmarks for our study - three SpecInt95 benchmarks, namely *compress, go* and *li*, two SpecFp95 benchmarks *su2cor* and *swim*, and *saxpy*, a toy benchmark.

5.2 Results

In the following subsections we present a detailed analysis of the results obtained. We first discuss the influence of standard optimizations on energy and power following which we study the effects of individual optimizations.

5.2.1 Influence of Standard Optimizations on Energy

Table 5.2.1 shows the results obtained when the benchmarks are compiled with different standard optimizations levels. We present the results of all optimizations relative to the result of optimization level `-O0`. For example, when we consider the number of instructions, the percentage of instructions executed by a benchmark optimized with option `-O2` is given by:

$$\% \text{ of Insts Executed by } Program_{O2} = \frac{\# \text{ of Insts Executed by } Program_{O2}}{\# \text{ of Insts Executed by } Program_{O0}} * 100$$

For example, in Table 5.2.1, we see that *compress* when compiled with `-O2` executed 17.96% fewer instructions than *compress* when compiled with `-O0`. Our results are presented in this form for all benchmarks and for all optimizations. As mentioned in Section 4, we used *cc* to compile the SpecInt benchmarks and *saxpy* and *g77* to compile the SpecFP benchmarks *su2cor* and *swim*.

We observe that the number of instructions committed drops drastically from optimization `-O0` to `-O1`, and also drops significantly in codes optimized with `-O2` and `-O3`. There is however a very marginal increase in the number of instructions in *compress*. In codes optimized with `-O4` option, the number of instructions increases due to the extra NOPs code generated for scheduling.

The reduction in number of instructions directly influences execution time or performance. The performance improvement is significant in `-O1` when compared to `-O0`, sometimes as high as 73% (*swim*). `-O2, -O3` also lead to significant improvement over `-O1`, for example, we see an 8% improvement in *li* with `-O2` optimization. In some benchmarks like *saxpy* the improvement is only about 0.6%. Optimizations `-O2, -O3` improve performance in *compress* even though the number of instructions increases.

The energy consumed by the code is again directly proportional to the number of instructions. Here we see that even though `-O2, -O3` improve performance in *compress*, the energy consumed is higher. This is because of the higher number of instructions. Hence, the amount of work done is more. In all the benchmarks, we see that the energy decreases when the number of instructions decrease. Hence, if we are compiling for energy, we should chose optimizations such as common sub-expression elimination, induction variable elimination and unrolling that reduce the number of instructions executed. Optimizations such as the ones in `-O4` (inserting NOPs to improve scheduling), may improve performance, but can also increase the number of instructions, leading to higher energy requirements. The energy increase is seen to be up to 4% (in *compress*).

5.2.2 Influence of Standard Optimizations on Power

To study the influence of compiler optimizations on power, we again refer to Table 5.2.1. We see that though the number of instructions and the number of cycles taken reduces in higher optimization levels, the number of instructions do not reduce enough to keep the instructions per cycle (IPC) constant. IPC reduces in `-O1` codes but increases in `-O2, -O3` and `-O4` codes. IPC in `-O0` is low because of the poor quality of code produced. Since optimizations such as common subexpression elimination improve code by reducing instructions rather than increasing available parallelism, IPC does not increase in `-O1` codes. Most optimizations that increase IPC such as instruction scheduling, loop unrolling etc are included in `-O2, -O3` and `-O4` levels. Power dissipated is the amount of work done in one cycle. This is directly proportional to the IPC. Hence, we see that optimizations that increase IPC, increase the power dissipated. Instruction scheduling and other `-O2, -O3` optimizations are good for performance improvement but are bad when instantaneous power is the main concern.

5.2.3 Influence of Individual Optimizations on Energy and Power

We refer to Tables 2 to 7 for experiments on how the different individual optimizations affect power/energy. We show the results for each benchmark separately. The tables show the performance, power and energy of each of the optimizations relative to performance, power and energy of code with `-O0` (similar to Table 5.2.1). Since the individual optimizations are applied over the `-O1` option, in our discussions, we always compare results of the optimizations with results of `-O1`. We first discuss the effects of the instruction scheduling options.

The *-fschedule-instr* optimization does simple basic block list-scheduling and *-fschedule-instr2* does aggressive global scheduling. We expect both options

Table 1: Effects of Standard Optimization on Power/Energy

Benchmark	opt level	Energy	Exec Time	Insts	Avg Power	IPC
compress	O0	100.00	100.00	100.00	100.00	100.00
	O1	74.48	81.55	81.52	91.33	99.96
	O2	75.13	81.44	82.04	92.25	100.73
	O3	75.13	81.44	82.04	92.25	100.73
	O4	79.01	82.77	86.11	95.45	104.03
go	O0	100.00	100.00	100.00	100.00	100.00
	O1	66.20	64.13	68.94	103.23	107.50
	O2	62.62	61.31	63.01	102.14	102.78
	O3	62.62	61.31	63.01	102.14	102.78
	O4	63.67	62.19	63.75	102.38	102.51
li	O0	100.00	100.00	100.00	100.00	100.00
	O1	81.32	83.66	83.18	97.20	99.42
	O2	79.60	75.97	82.97	104.78	109.21
	O3	79.60	75.97	82.97	104.78	109.21
	O4	85.71	77.89	90.96	110.05	116.78
saxpy	O0	100.00	100.00	100.00	100.00	100.00
	O1	97.38	100.24	92.49	97.15	92.27
	O2	97.69	99.38	92.49	98.30	93.07
	O3	97.69	99.38	92.49	98.30	93.07
	O4	98.31	99.27	92.84	99.02	93.51
su2cor	O0	100.00	100.00	100.00	100.00	100.00
	O1	42.09	51.04	33.21	82.46	65.06
	O2	40.99	47.52	33.10	86.28	69.67
	O3	40.99	46.37	33.10	87.65	71.38
swim	O0	100.00	100.00	100.00	100.00	100.00
	O1	30.10	36.64	20.01	82.15	54.63
	O2	28.93	34.01	19.05	85.06	56.01
	O3	28.93	34.01	19.05	85.06	56.01

Table 2: Individual Optimizations on Compress

opt level	Energy	Exec Time	Insts	Power	IPC
O0	100.0	100.0	100.0	100.0	100.0
O1	67.66	74.68	60.46	90.60	80.95
inline-func	67.69	74.68	60.46	90.63	80.95
sched-instr2	68.82	74.94	63.21	91.82	84.35
sched-instr	66.66	73.47	59.83	90.72	81.43
unroll-loops	66.84	74.19	59.90	90.09	80.74

Table 3: Individual Optimizations on li

opt level	Energy	Exec Time	Insts	Power	IPC
O0	100.00	100.00	100.00	100.00	100.00
O1	70.91	74.67	66.18	94.96	88.63
inline-func	71.02	73.14	68.00	97.11	92.97
sched-instr2	69.56	66.65	68.33	104.36	102.52
sched-instr	69.56	66.65	68.33	104.36	102.52
unroll-loops	66.05	59.91	68.19	110.24	113.81

to increase the IPC and hence the power. We can see from the tables that IPC goes up in most benchmarks, in some benchmarks up to 4.6% (in *su2cor*). The power increase is up to 3.9% . In *li*, the power increases by as much as 10%. The aggressive scheduler (prepass scheduler) increases register pressure

Table 4: Individual Optimizations on saxpy

opt level	Energy	Exec Time	Insts	Power	IPC
O0	100.00	100.00	100.00	100.00	100.00
O1	96.78	98.56	96.21	98.19	97.61
inline-func	96.78	98.56	96.21	98.19	97.61
sched-instr2	97.07	97.14	96.27	99.93	99.11
sched-instr	96.79	98.52	96.15	98.24	97.60
unroll-loops	96.87	98.72	95.97	98.13	97.21

Table 5: Individual Optimizations on su2cor

opt level	Energy	Exec Time	Insts	Power	IPC
O0	100.00	100.00	100.00	100.00	100.00
O1	42.09	51.04	33.21	82.47	65.07
inline-func	42.06	51.01	33.21	82.46	65.11
sched-instr2	42.49	50.36	34.02	84.38	67.55
sched-instr	40.90	47.79	33.30	85.58	69.67
unroll-loops	40.17	48.35	31.17	83.08	64.46

Table 6: Individual Optimizations on swim

opt level	Energy	Exec Time	Insts	Power	IPC
O0	100.00	100.00	100.00	100.00	100.00
O1	30.06	36.64	20.02	82.02	54.64
inline-func	30.06	36.64	20.02	82.02	54.64
sched-instr2	30.91	36.39	20.53	84.92	56.41
sched-instr	29.83	35.11	20.32	84.95	57.86
unroll-loops	29.29	35.38	18.19	82.80	51.43

Table 7: Individual Optimizations on go

opt level	Energy	Exec Time	Insts	Power	IPC
O0	100.00	100.00	100.00	100.00	100.00
O1	40.97	42.75	42.65	95.83	99.77
inline-func	40.92	42.78	42.58	95.64	99.54
sched-instr2	43.07	44.01	45.25	97.87	102.82
sched-instr	43.52	44.89	46.52	96.96	103.63
unroll-loops	39.38	41.95	39.30	93.88	93.69

and hence causes significant number of spills, thereby increasing the total number of instructions executed and the total energy. The increase in number of instructions and energy are up to 3.52% and 2.14% respectively. This optimization needs to be improved upon if power and energy are a concern. We would see a greater impact of these optimizations if the target processor was an in-order machine, wherein the compiler if fully responsible for exposing the parallelism. In an out-of-order issue machine, the hardware can find the parallelism even if the compiler does not do any reordering. The reason why we see some improvement in performance (and increase in IPC) is because the hardware is limited by the instruction window size, the global scheduler which has the full program as its scope helps the hardware see more instructions than it otherwise would have.

We next discuss the impact of unrolling. Unrolling appears to be a good optimization to use for energy because the number of instructions reduce significantly. We are able to reduce the number of instructions by 3.35% in *go*, the energy falls by 1%. We see that in the some benchmarks the energy falls by 5% (*li*). However, reducing the energy does not necessarily reduce power. For instance, in *li*, the power goes up by 10%. Unrolling increases the size of the basic block, hence allows the hardware increase the overlap of instructions. This leads to an increase in the number of simultaneous operations being executed. It may be noted that the IPC in *li* increases by 25%. However, this observation is not consistent among all the benchmarks, in many benchmarks, there is no increase in IPC. This is because the target architecture has a good branch predictor, it does unrolling in hardware, hence reducing the impact of software unrolling. We are currently investigating how the unrolling optimization affects power if we turned off the branch prediction hardware. We expect to see a significant increase in IPC and power in the codes after unrolling has been applied.

Our next optimization is inlining of function calls. Inlining as explained in the motivation section will reduce the number of instructions and hence energy. However, in our benchmarks, only *go* and *su2cor* a very marginal decrease in energy. In our future work, we will be investigating further with a better set of benchmarks more suited for this optimization.

6 Conclusions

In this paper we evaluated the impact of using the different levels of optimizations in the *cc* compiler on system power and energy. We also evaluated the effect of a few individual optimizations. We found the that energy consumption reduces when the optimizations reduce the number of instructions executed by the program, i.e., when the amount of work done is less. The standard optimization level `-O1` reduces the number of instructions drastically as compared to `-O0` because it invokes optimizations such as common subexpression elimination, an optimization used to eliminate redundant computations in the program. The drop is not as that significant in `-O2, -O3 and -O4` optimizations. The energy also drops in the same proportion.

We found power dissipation to be directly proportional to the average IPC of program. `-O2, -O3 and -O4` levels have significantly higher IPC and hence higher average power. The optimization levels `-O2, -O3 and -O4` include optimizations such as instruction scheduling, which are typically used to increase the parallelism in the code.

Out of the four individual optimizations we evaluated, we found unrolling to be a good optimization for energy reduction but it increases power dissipation. Function inlining is good for both energy reduction and reducing power dissipation. Instruction scheduling was found to be a bad optimization to use when power is a concern. Simple schedulers did not affect the energy consumption, but aggressive schedulers i.e., schedulers that increased register pressure and introduced spills, increased the energy consumption as well. For our future work, we would like to evaluate more individual optimizations and improve the ones that we find are currently unoptimized for power or energy.

References

[1] K. Asanovic. Energy-exposed instruction set architectures. *Work In Progress Session, Sixth International Symposium on High Performance Computer Architecture*, Jan 2000.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th International Symposium on Computer Architecture*, Jun 2000.

[3] J. Bunda, W. C. Athas, and D. Fussell. Evaluating power implication of cmos microprocessor design decisions. In *1994 International Workshop on Low Power Design*, April 1994.

[4] D. Burger and T. M. Austin. Evaluating future microprocessors: The simplescalar tool set. Technical report, Dep. of Comp. Sci., Univ. of Wisconsin, Madison, 1997.

[5] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *Design Automation Conference*, pages 726–731, 1998.

[6] N. B. I. Hajj, C. Polychronopoulos, and G. Stamoulis. Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors. In *ISLPED 98*, pages 70–75, July 1998.

[7] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Design Automation Conference*, July 2000.

[8] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *30th International Symposium on Microarchitecture*, pages 184–193, Dec 1997.

[9] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *25th International Symposium on Computer Architecture*, pages 1–10, Jun 1998.

[10] C.-L. Su and A. M. Despain. Cache designs for energy efficiency. In *28th Annual Hawaii International Conference on System Sciences*, pages 306–315, 1995.

[11] C. L. Su, C. Y. Tsui, and A. M. Despain. Low power architecture design and compilation techniques for high-performance processors. In *IEEE COMPCON*, Feb. 1994.

[12] M. C. Toburen, T. M. Conte, and M. Reilly. Instruction scheduling for low power dissipation in high performance microprocessors. In *Power-Driven Microarchitecture Workshop In Conjunction With ISCA 1998*, Jun 1998.