Understanding the Energy-Delay Tradeoff of ILP-based Compilation Techniques on a VLIW Architecture

Gilles Pokam
IRISA
Campus Universitaire de Beaulieu
35042, Rennes, Cedex France

François Bodin
IRISA
Campus Universitaire de Beaulieu
35042, Rennes, Cedex France

Abstract

Energy consumption is becoming an important issue on modern processors, especially on embedded systems. While many architectural solutions to reduce energy consumption exist, software solutions on the other hand mostly rely on performance optimization techniques to address that problem. The rationale behind this latter approach is based on the unstated rule that energy consumption is roughly proportional to the execution time. While some ILP techniques allow to increase performance by eliminating redundant instructions, others however do increase the total instruction count, mitigating their benefit as far as energy consumption is concerned.

This paper explores the energy-delay tradeoff of ILP enhancing techniques at the compilation level. The goal is to understand the main energy issues involved by forming ILP blocks. We present an analytical methodology which essentially exploits the variations in program performance to identify conditions leading to energy consumption increase. Our results show that there exists a threshold above which ILP enhancing optimizations may necessarily turn into diminishing energy reduction returns. The proposed tradeoff analysis reveals that this can be mainly attributed to the limited available instruction parallelism of applications which causes wasted computation and, to some extent, machine overhead to start dominating the energy consumption in some scenarios where the ILP is pushed above a given threshold.

1 Introduction

Modern processors heavily rely on instruction level parallelism (ILP) techniques to achieve high performance. With statically scheduled VLIW processors, this is capitalized by the duplication of functional units that provide support for the concurrent execution of multiple instructions. On these machines however, the main challenge essentially lies on the compiler which is in charge of finding enough instructions to fully utilize the underlying hardware parallelism. For this purpose, many optimizing VLIW compilers are built with sophisticated ILP transformation techniques such as superblocks [7], trace scheduling [5], and hyperblocks [10].

These techniques are based on the principle that a larger scheduling scope unit is formed as the result of merging several basic blocks into a single one. While these techniques indubitably improve ILP, their impact on the energy consumption is mitigated by the increase in the total instruction count they usually cause. On embedded systems where a large portion of the dissipated energy emanates from the running programs [16], an increase in the energy consumption can have a dramatic impact on the system energy efficiency.

Common approaches used to tackle these issues at the software level heavily rely on performance optimization, following an unstated rule that energy consumption is roughly proportional to the total execution time. This however neglects the impacts due to computational and architectural overhead, which mainly result from wasted computation and the diminishing performance returns of incrementally applying some optimizations.

This paper explores the energy-delay tradeoff of applying ILP optimization techniques on a statically scheduled VLIW processor. Our goal is to develop a theoretical understanding of the main energy issues involved by ILP transformation techniques. To this aim, we developed an analytical energy-efficiency model to investigate the issues between energy and performance. The model capitalizes on the fact that monitoring the variations in program performance can be achieved on many modern processors through some form of prediction mechanism or profiling at the software level. We exploit this approach to show that there exists a threshold above which optimizing for ILP may turn into diminishing energy reduction returns. We translated these results into heuristics to measure the energy-efficiency of a ILP transformation on a set of embedded applications.

The remainder of this paper is organized as follows. In Section 2, we define the metric we used throughout this study. In Section 3, the energy model of a VLIW-core processor is introduced, followed by the analytical model for the energy-efficiency. In Section 4, an hyperblock formation case study is presented and evaluated. Section 5 discusses previous work and Section 6 concludes.

2 Evaluation Metric

We consider the energy-delay product [6] to define another equivalent metric; the ratio performance-to-energy (PTE) that we use to compare two different instances of an application at the software level. In comparing instances of the same application, the PTE ratio attempts to measure the need of increase performance against low energy. The PTE ratio is expressed as the reverse of the energy-delay product as shown in (1).

$$PTE = \frac{1}{E_{op} \times Cycle_{op}} = \frac{Performance}{Energy} \tag{1}$$

The advantage of the PTE ratio expressed above is that, given an energy budget, one can look at different performance values that potentially improve the PTE ratio without worsening the energy. In this way, we can focus on the range of values that optimize the energy-delay product with a given target. This approach can serve compilers since it does not relies on special hardware supports. We only need to track the changes in processor performance. This however can be mastered in a relatively easy manner at the software level.

3 Energy-Efficiency Analysis

This section first introduces the VLIW-core energy model that serves us as a foundation for the energy-delay tradeoff exploration. Then, at the end of this section, we discuss the theoretical model developed to allow compile-time energy-delay evaluation of the applied ILP transformation.

3.1 Energy model

We use a VLIW-core energy model [2] that features an embedded VLIW processor capable of issuing up to N parallel operations in a very long instruction called a bundle. In this model, the energy, EPB, dissipated by the execution of a bundle w_n is modeled as follows:

$$EPB_{w_n} = E_c + IPC_{w_n} \cdot E_{op} + m \cdot p \cdot E_s + l \cdot q \cdot E_{miss} \tag{2}$$

In (2), E_c refers to a constant average energy base cost, IPC_{w_n} is the number of operations different from NOP in the bundle, and E_{op} is the average energy consumption associated with the execution of an operation. Note that, in contrast to superscalar processors, a VLIW processor does not provide aggressive architectural features [13] such as rename and wakeup logic, which otherwise should have largely contributed to increase the energy consumption per operation. In such a case, we could expect the average energy consumed per operation (E_{op}) in a VLIW processor to be roughly the same for almost all operations, not counting operations involving a memory access. This is however not exactly true, since the circuit state effect [16] due to the switching activity on the bus may make the energy base cost of one instruction higher than expected.

The third expression in (2) refers to the additive energy consumption due to a miss event on the D-cache, where m is the average number of additional stall cycles per bundle due to a D-cache miss, p the probability that a D-cache miss occurs, and E_s the core energy consumption during a pipeline stall. The fourth expression is related to the I-cache misses, where l is the average number of additional NOP operations per bundle introduced during a I-cache miss, q the probability per bundle that this event occurs, and E_{miss} the energy consumption of the core during an I-cache miss.

Some dynamic program behaviors can not be easily captured at compile time. This includes for instance the data or instruction cache miss count. In order to simplify the above model without losing to much accuracy, we restricted the study on computation-intensive program regions, e.g loops. The above model can therefore be simplified, namely by neglecting the impact due to the I-cache misses. This consideration is indeed true as long as the undertaken ILP transformations do not increase the size of the loop region in such a way that it overwhelms the I-cache. The energy model described in (2) can then be rewritten as follows:

$$EPB_{w_n} = E_c + IPC_{w_n} \cdot E_{op} + m \cdot p \cdot E_s \tag{3}$$

We also assume for this study that the presumed machine model does no prefetching. This is necessary to exclude the unpredictable effects of prefetching on the data cache which can impact performance and energy, thereby biasing the analysis. On some machine, a compiler option is sometimes provided to disable prefetching. We assume this is the case for the rest of this study.

3.2 Energy-efficiency cost model

In order to estimate the energy consumption impact for an ILP transformation at the compilation level, we express the PTE ratio as a function of both the energy and the performance. For the rest of this study, we consider the smallest scheduling scope granularity to be the basic block. We then obtain an expression for PTE as shown in (4).

$$PTE = \frac{1}{E_{BB} \times Cycle_{BB}} = \frac{\overline{IPC}}{N \times E_{BB}} \tag{4}$$

N and $\overline{IPC}$ represent the number of operations contained in the basic block and the average number of operations executed in each bundle respectively. The basic block energy term E_{BB} in (4) can be decomposed as the sum of the energy required to execute an instruction stream of n consecutive bundles. This can formally be expressed as follows:

$$\begin{aligned} E_{BB} &= f \cdot \sum_{i=1}^{n} EPB_i \\ &= f \cdot n \cdot (E_c + E_{op} \cdot \overline{IPC}) + \overline{s} \cdot E_s \end{aligned} \tag{5}$$

In (5), $\overline{s}$ is the average number of stall cycles due to data cache miss in the basic block, whereas f represents the basic block execution frequency. If we consider a coarser granularity, the energy consumption of a region R composed of m basic blocks can be modeled in a similar manner, as shown in (6).

$$\begin{aligned} E_R &= \sum_{i=1}^{m} E_{BB_i} \\ &= m \cdot \overline{f_R} \cdot \overline{n_R} \cdot (E_c + E_{op} \cdot \overline{IPC_R}) + \overline{s_R} \cdot E_s \end{aligned} \tag{6}$$

In (6), $\overline{f_R}$ and $\overline{n_R}$ are the average execution frequency and the average number of bundles of the region R respectively.

Recall that the rationale behind the PTE ratio is to lay emphasis on the range of performance values that improve the energy efficiency. This is necessary to compare program instances corresponding to the states before and after an ILP transformation. Therefore, the IPC term of the PTE ratio can be viewed as a kind of gear mechanism for energy regulation purpose. In this way, we can keep track of the IPC values that improve the energy efficiency and then only select those candidate regions that lead to this IPC improvement. Then, given a candidate region R composed of m basic blocks, we call a particular ILP transformation function, F_{ilp}, energy efficient if the PTE ratio of the resulting ILP block H is such that $PTE_H > PTE_R$. The inequality can be solved at $\overline{IPC_H}$ to yield a new inequality shown in (7).

$$\overline{IPC_H} > F_{ilp}(\overline{IPC_R}) \tag{7}$$

The expression for F_{ilp} is equivalent to

$$F_{ilp}(\overline{IPC_R}) = \frac{A \cdot \overline{IPC_R}}{B + C \cdot \overline{IPC_R}} \tag{8}$$

with

$$\begin{cases} A = f_H \cdot N_H \cdot n_H \cdot E_c + N_H \cdot \overline{s_H} \cdot E_s \\ B = m \cdot N_R \cdot \overline{f_R} \cdot \overline{n_R} \cdot E_c + N_R \cdot \overline{s_R} \cdot E_s \\ C = (m \cdot N_R \cdot \overline{f_R} \cdot \overline{n_R} - f_H \cdot N_H \cdot n_H) \cdot E_{op} \end{cases} \tag{9}$$

The process of evaluating the above transformation function requires to characterize the candidate region R as well as the target ILP block H, as indicated by the substitution variables B and A of the function F_{ilp}. This latter variable is however not obvious to determine, since it requires that we know in advance what the target ILP block H will look like. This is mainly due to the fact that several transformations may be applied on the resulting ILP block as a side-effect of some ILP-based optimizations, causing the number of bundles and executed operations to vary. At this stage, we therefore anticipated the formation of the ILP block by integrating a region ahead scheduling pass to predict the characteristics of the target ILP block, yielding a rough indication of the ILP block parameters.

3.3 Tradeoffs analysis

We have studied the energy-delay tradeoff in terms of the amount of ILP needed to compensate for an increase in the energy dissipation. The increase in energy is measured as the additional overhead and/or wasted energy. By wasted energy we mean essentially the energy dissipated due to the execution of needless operations. Such operations result from if-converting basic blocks along the taken and the not-taken paths of a branch instruction. These operations consume extra resources since the processor still have to fetch, decode and execute them, although no machine state change is guaranteed to occur. In the ILP transformation function F_{ilp}, the wasted energy can be termed by the variable C. As one can observe, the wasted energy is proportional to the product of the average energy per operation and the dynamic operation count difference between the candidate region and the target ILP block. This operation count difference constitutes the extra operations executed by the processor. We can analyze the effect due to the execution of these extra-instructions under 3 different aspects.

Case $C > 0$. In this case, we can expect the amount of wasted energy to be negligible, because the executed extra work is marginal. Obviously, this case corresponds to the optimal execution scenario. Typically, the condition $C > 0$ is true when the value of the ILP block term is quantitatively less than that of its corresponding original schedule. This may be primarily a consequence of the fact that applying some types of optimization on the resulted ILP block may produce more impact in terms of reduction of the number of operations N_H and static scheduled length n_H compared to its original schedule. Examples of such optimization may include instruction merging and instruction renaming which respectively reduces the number of operations N_H and diminishes the

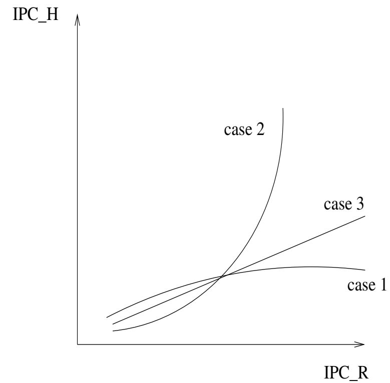

Figure 1: Shape of the curves corresponding to the cases $C < 0$, $C = 0$, and $C > 0$ of the inequality (7) for various values of IPC_H and IPC_R.

impact of dependence height by allowing instructions to be scheduled earlier, thereby reducing the static scheduled length n_H. The class of energy-efficient solutions for which the transformed block yields the best energy-delay tradeoff lies above the curve labeled $case1$ in Figure 1. In this figure, it can be seen that the curve has a logarithmic shape. This indicates that the probability to get some values of $\overline{IPC_R}$ that will feet well to the inequality (7), providing lower values for N_H and n_H, is high.

Case $C = 0$. In some extent, this case can be assumed to be similar to the previous one because there is no additional extra computation than what is required for the normal computation. However, at equal schedule length and/or operation count, deciding on the energy-efficiency of the transformation is not as straightforward as it seems to be. Some energy side effects may render the ILP block less energy-efficient than its original schedule. A typical situation is when the circuit effect in the transformed ILP block is no longer negligible. This may be principally due to the inter-instruction effects in the new resulting schedule. If the compiler can evaluate this impact, the decision can be made simpler. If we consider this circuit effect to represent a given fraction of the ILP block energy, we can then expect the set of solutions for which the ILP transformation function yields a good energy-delay tradeoff to be at a corresponding distance above the curve labeled *case 3* shown in Figure 1.

Case $C < 0$. In this case, the amount of wasted energy is no longer negligible because the extra computation cost start to be the dominant factor. In the variable C, this is reflected by the fact that the value of the ILP block term is quantitatively higher than that of its corresponding original schedule. In another words, it means that the values of N_H and n_H increase non-linearly with respect to their corresponding values in the original schedule. There are many scenarios which can potentially lead to degrade the values of N_H and/or n_H. Consider the case of a dependence height mismatch. The static scheduled length n_H may increase as a result of lengthening the execution time of the taken path, thereby offsetting the ILP benefit by degrading the average energy consumption of that path. In a intensively executed loop region, this may not be negligible at all. Consider again a machine with scarce resources, e.g one memory port as in our machine model. The resulting schedule length n_H may also increase if resource contention becomes an important issue when paths are overlapped. On the other hand, the increase of the instruction count N_H is more concerned with the amount of compensation code that is introduced in order to repair the effect of the elimination of branch instructions. These effects, combined with the fact that applications usually show not enough available parallelism [12], can rather turn the ILP benefit into additive computation cost and therefore increase energy consumption. This scenario is illustrated in Figure 1, in the curve labeled *case 2*. The exponential shape of the curve demonstrates that the range of energy-efficient solutions which satisfies the inequality (7) is quite small because IPC_H grows exponentially with increasing IPC_R. This means that, for an ILP transformation to be energy-efficient given a value of $\overline{IPC_R}$, the transformation function should yield a value for $\overline{IPC_H}$ which is above the aforementioned curve.

The energy overhead is principally due to the processor core activity, independently of the execution context of a bundle. It is therefore directly related to the architectural implementation cost of the processor. Because of its tight relation with the processor implementation, the energy overhead is difficult to evaluate without a detailed architectural-level simulator. We can however have a theoretic rough estimate of its impact if we again consider the transformation function F_{ilp}. Under optimal data cache conditions, the curve of the ILP transformation function has an asymptote which is proportional to the ratio $\frac{E_c}{E_{op}}$, as shown in (10).

$$\lim_{\overline{IPC_R}\to\infty}\left(\frac{A\cdot\overline{IPC_R}}{B+C\cdot\overline{IPC_R}}\right)=k\cdot\frac{E_c}{E_{op}} \tag{10}$$

If this ratio is made too large (higher E_c values), then the function quickly converges near the asymptote as IPC_R increases. Therefore, the range of energy-efficient solutions that satisfies the inequality (7) (values above the curve) is also expected to narrow as we gradually move towards larger IPC values. Smaller values for the ratio are therefore preferable in order to keep the range of feasible energy-efficient solutions within reasonable IPC values.

4 The Hyperblock Case Study

This section analyzes the energy issues involved by constructing hyperblocks. We first introduce the hyperblock framework model used throughout this study. Then, based on the previously discussed energy model, a set of heuristics is proposed to drive the formation of energy efficient hyperblocks. Finally, an evaluation of the proposed heuristics is provided at the end of this section.

4.1 The hyperblock framework model

We assume a guarded instruction model that relies on the use of a select instruction. The format of the select instruction is as shown below:

$$slctf\ dest = cond,\ src1,\ src2$$

The $slctf$ instruction writes the value of $src1$ into $dest$ if $cond$ is 0, otherwise $src2$ is written into $dest$ if $cond$ takes the value 1. In this model, the predication process goes through 2 steps. In the first step, the candidate basic blocks are selected for if-conversion. In the second step, the if-converted region is formed by eliminating the branch instructions and then inserting the

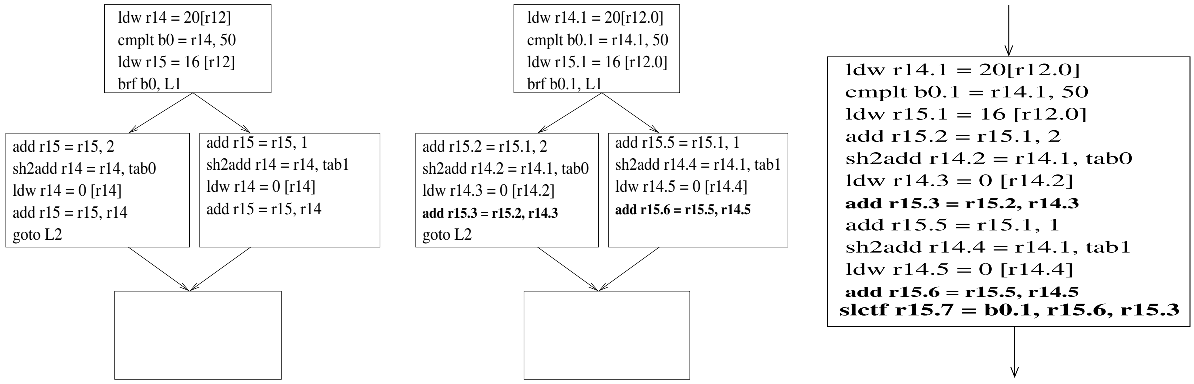

Figure 2: Original CFG (left), CFG in SSA (middle) and corresponding if-converted CFG (right).

appropriate select instructions into the code whenever there are more than one definition of the same register that flow into the resulted if-converted region. This last step puts a strong emphasis on the register names, requiring that each register be uniquely named throughout the program. Prior to if-conversion, the original CFG is therefore transformed into a SSA form to guarantee the uniqueness of each register name. This is illustrated in Figure 2.

The limited predication scheme presented above differs from the one presented in [10] in that we do not allow an hyperblock to contain multiple exit points or branches. Therefore, the regions to be considered for inclusion in a hyperblock consist solely of set of consecutive basic blocks or hammocks. An hyperblock in this framework is thus akin a large basic block of predicated instructions with single entry and exit points.

4.2 Energy heuristics

We consider the inequality (7) as the main heuristic for deciding whether or not to transform a set of candidate blocks into a hyperblock. In order to address the cost-effectiveness of applying such a transformation, we also consider a profit heuristic. We define the profit of performing a transformation as follows:

$$Profit = \frac{PTE_{transformed} - PTE_{original}}{PTE_{original}} \tag{11}$$

The profit is used to weight the gain of the transformation. To this end, the profit is compared against an arbitrary threshold value which puts a minimal bound on the potential gain required to effectively complete the transformation. The hyperblock transformation process traverses the n-odes of the loop tree in postorder to insure that innermost loops are processed first. Then, for each loop, candidate regions for if-conversion are selected and sorted from the innermost to the out-ermost to guarantee that no region is transformed that contains a nested non-if-converted region. Thereon, each region is checked against the if-conversion test, i.e the inequality (7) in the model. This step is preceded by the region ahead scheduling pass from which the target hyperblock characteristics are extracted. The profit of the transformation is then evaluated against the threshold value if the if-conversion test succeeds. In order to capture the global effect of the transformation on the program, the profit is also evaluated at the CFG level. Each region is then annotated with its potential gain. The region that exhibits the highest gain is selected for if-conversion. The CFG is then immediately updated to reflect this new change and the profit for each region is anew computed until no more change occurs. This guarantees that the performance and the energy are globally well balanced.

4.3 Energy characterization on a VLIW processor

In the following subsections, we present an evaluation of the hyperblock formation heuristics described above. We first introduce the evaluation platform and the simulation methodology used. Then, at the end of this section, we present and discuss our experimental results.

4.3.1 Platform and simulation methodology

Simulation platform

Our simulations were carried out on the Lx platform [4]. The Lx platform belongs to a family of customizable multi-cluster VLIW architectures. The implementation used in this study features a 4-issue width processor in which each cluster is composed of 4 ALUs, 2 multipliers, 1 Load/Store and 1 Branch unit. The register bank in each cluster includes a set of 64 32-bit general purpose registers and 8 1-bit branch registers to store the branch condition, the predicates and the carries.

Simulation methodology

The Lx platform is provided with a complete software tool-chain, where no visible changes are exposed to the programmer. The tool-chain includes, among other things, an aggressive ILP compiler, called the Lx compiler, which is derived from [9]. We used the Lx compiler to generate an input assembly prior that any aggressive ILP transformations are been performed onto the code and to run a Lx binary executable. This optimization level actually corresponds to the stage where all traditional scalar optimizations have been undertaken.

The extracted assembly code is then processed by SALTO [3] and ABSCISS [1]. SALTO is a general, compiler-independent tool that makes the manipulation of the assembly code at the CFG level easier. SALTO is also machine independent and can be parameterized to feature a specific target by providing it with a machine description file. We use SALTO essentially to modify the input assembly code by adding new compilation passes. In particular, we added two new passes: a hyperblock formation and optimization pass and a list-based scheduling pass that produces the final assembly code. Currently, the hyperblock optimization pass comprises instruction renaming, instruction merging and instruction promotion. Prior to SALTO, the input assembly code is first processed by ABSCISS, which is a SALTO-based compiled simulator. ABSCISS is mainly used to profile the assembly code at the basic block level and annotate it with runtime informations including the data miss count and the execution frequency.

Benchmarks	Description	region representativeness
adpcm coder	voice encoding	45%
adpcm decoder	voice decoding	43%
bffo	implementation of find first zero	59%
des	data encryption	38%
g3fax	fax decoding	33%

Table 1: Benchmarks description.

The machine specific simulation parameters E_c, E_{op}, and E_s were obtained directly from [2], where the authors also provided a validation of the energy model used in this paper for a simulation platform that is similar to ours. According to the authors [2], one must count with an error range that approaches an average of 5.2% compared to the real RTL power estimates.

4.3.2 Results

We realized our experiments on a subset of the Powerstone benchmarks [15]. The selected benchmarks were chosen according to the amount of hyperblock formation opportunity found into them. The characteristics of the selected benchmarks are shown in Table 1. The last column in the table gives the percentage of basic blocks contained in the candidate regions compared to the total number of basic blocks in the CFG.

The energy-delay values are presented in Figure 3. Each bar in the figure corresponds to the original CFG, the CFG with the hyperblock formation heuristics and the CFG in normal hyperblock form, respectively. We can observe from the figure that, in almost all cases at the exception of the last one, the energy-delay product has a better energy-efficiency when the energy heuristics are integrated as part of the hyperblock formation process. On average, we observe an improvement of the system energy-efficiency of more than 17%, which, given the same power consumption budget, practically signifies that the autonomy period of a battery-powered system is lengthened.

These energy-delay values can be better understood by looking at the other results presented in the same figure. In this figure, we plotted the values corresponding to the static schedule length, i.e the number of cycles not counting the stall cycles due to the Dcache and Icache misses, the total operation count and the resulted IPC. The static schedule length and the total operation count aim at estimating the performance cost of transforming a region into a hyperblock. We can indeed observe from the Figure 4 and Figure 5 that the energy-delay value of the hyperblock scheme is closely correlated with the increase of the static schedule length and the number of executed operations. When the static schedule length and the operation count of the hyperblock scheme are higher than their values in the scheme with the integrated energy heuristics, the corresponding energy-delay product is also not better. This is especially the case when the corresponding increase in the IPC is not large enough to compensate for this, as shown in Figure 6 (bottom right figure). We observe this phenomenon by the adpcm coder, the adpcm decoder, the bffo and the des benchmarks where the IPC improvement only averages 9% for a total of 15% degradation of the static schedule length and less than 6% increase of the total operation count. The main reasons why the static schedule length and the total operation count increase are due to the effect of the

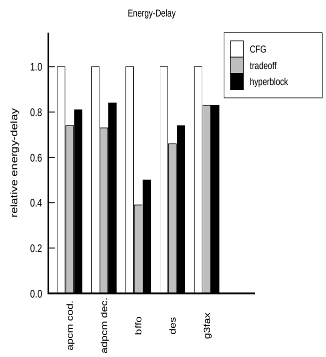

Figure 3: Energy-delay.

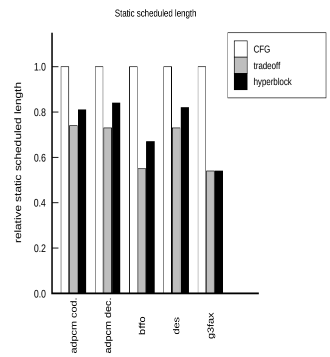

Figure 4: Static scheduled length.

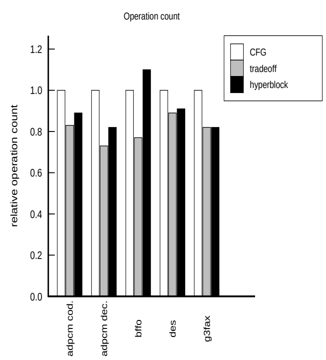

Figure 5: Operation count.

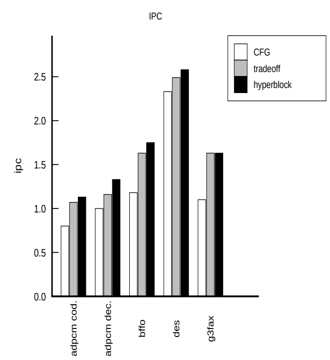

Figure 6: IPC.

dependence height mismatch and resource contention on the one side, and the insertion of the select instructions on the other side.

When the ILP transformation does not considerably degrade the static schedule length and the total operation count, the energy heuristics act transparently to the normal hyperblock formation process. This explains why the energy-delay values of the last benchmark are nearly the same in both scenarios. Definitively, an integrated energy heuristic can be used concurrently to the ILP formation process to prevent the compiler from doing optimization when the application has reached a given IPC threshold. This offers the opportunity to discriminate among optimizations that stress the ILP from those that do the same, but in a less efficient manner.

5 Related Works

There have been tremendous works that have addressed the issues of reducing the energy/power consumption on general purpose or embedded systems. Most of these works have however principally benefit the computer designer community. Until recently, many researches have started focusing on software optimization techniques to provide an alternative solution to that problem.

In [14], the authors present a study of the impact of instruction scheduling on the energy and performance. In this study, the authors highlight the existing tradeoff between performance and energy by experiencing several list-based scheduling algorithms. They arrive at the conclusion that most scheduling heuristics are not energy-efficient and need therefore to be revisited. The methodology used consisted in scheduling instructions in a DAG based on a energy cost table similar to the model proposed in [17]. Similar work on instruction scheduling include the study by Lee et al. [8] which reduces the total power dissipated by minimizing the switching activity on the bus.

Other works which are closely related to ours include the study by [11] and [18]. In [11] for instance, the author proposes to rely on code profiling to annotate program instructions that can be executed concurrently, hoping in this way to reduce the activity on the fetch issue unit. The approach in [18] follows the same goal, but proposes instead to rely on compiler-based IPC prediction to reduce the activity on the fetch unit. In both cases, the energy consumption can be reduced if the activity on the fetch issue unit can be decreased, because few functional units will be exercised in this way. Our approach is to some extent similar to these two works because we also indirectly seek at reducing the activity on the fetch unit by trading the IPC for energy reduction. However, this can be seen as a "bonus" to our approach since our primarily intention is to enable/disable the application of aggressive ILP optimization techniques whenever this can result to a better energy-delay product. These approaches are therefore complementary each one to each other.

6 Conclusions

A theoretical framework for the energy-delay analysis of ILP transformation techniques on a VLIW-based embedded system has been proposed and experimented. The main purpose of this study was to point out that it is rather questionable if applying aggressive ILP optimization techniques results into an improved energy-efficient system. In particular, we have made clear that aggressive ILP optimization techniques, in addition to improving the ILP, may also emphasize the execution of needless operations, making the benefit obtained by the first to be cancelled as a result of the increasing wasted energy consumption due to the second. Our results have shown that up to 17% energy-delay improvement can be achieved by a compiler that is aware of this. This also comforted our conviction that exposing architecture-based energy features to the compiler may help improving the overall energy efficiency, especially for embedded systems. We plan to experiment the proposed technique with more robust compiler and extend the range of ILP transformation techniques to include superblocks and trace scheduling.

References

[1] Amicel, R., and Bodin, F. A new system for high-performance cycle-accurate compiled simulation. In *Proceedings of the 5th International Workshop on Software and Compilers for Embedded Systems*, 2001.

[2] Benini, L., Bruni, D., Chinosi, M., Silvano, C., Zaccaria, V., and Zafalon, R. A power modeling and estimation framework for vliw-based embedded systems. In *Proceedings of the IEEE Eleventh*

International Workshop on Power and Timing Modeling, Optimization and Simulation, September 2001.

[3] Bodin, F., Rohou, E., and Seznec, A. Salto: System for assembly-language transformation and optimization. In *Proceedings of the Sixth Workshop on Compilers for Parallel Computers*, 1996 December.

[4] Faraboschi, P., Brown, G., Fisher, J.A., Desoli, G., and Homewood, F. Lx: A technology platform for customizable vliw embedded processing. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[5] Fisher, J.A. Trace scheduling: A technique for global microcode compaction. *IEEE Transaction on Computers*, 30(7):478–490, 1981.

[6] Gonzalez, R. and Horowitz, M. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1283, September 1996.

[7] Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G, Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., and Lavery, D.M. The superblock: An effective technique for vliw and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, 1993.

[8] Lee, C., Lee, J.K., and Hwang, T. Compiler optimization on instruction scheduling for low power. In *Proceedings of 13th International Symposium on System Synthesis*, September 2000.

[9] Lowney, P.G., Freudenberger, S.G., Karzes, T.J., Lightenstein, W.D., Nix, R.P., O'Donnell, J.S., and Ruttenberger, J.C. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[10] Mahlke, S. A., Lin, D. C., Chen, W. Y., Hank, R. E., and Bringmann, R. A. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.

[11] Marculescu, D. Profile-driven code execution for low power dissipation. In *Proceedings of ACM International Symposium on Low Power Design (ISLPED)*, July 2000.

[12] Nicolau, A., and Fisher, J. Measuring the parallelism available for very long instruction word architectures. *IEEE Transaction on Computers*, 33(11):968–976, November 1984.

[13] Palacharla, S., Jouppi, N., and Smith, J.E. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual Internacional Symposium on Computer Architecture*, pages 206–218, June 1997.

[14] Parikh, A., Kandemir, M., Vijaykrishman, N., and Irwin, M.J. Instruction scheduling based on energy and performance constraints. In *Proceedings of the IEEE Computer Society Annual Workshop on VLSI*, April 2000.

[15] Scott, J., Lea Hwang Lee, John Arends and William Moyer. Designing the low-power m.core architecture. In *Proceedings of Power Driven Microarchitecture*, June 1998.

[16] Tiwari, V., Malik, S., and Wolfe, A. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2(4):437–445, December 1994.

[17] Tiwari, V., Malik, S., Wolfe, A., and Lee, M. Instruction level power analysis and optiomization of software. *Journal of VLSI Signal Processing Systems*, (1):1–18, 1996.

[18] Unsal, O.S., Koren, I., Krishna, C.M., and Moritz, C.A. Cool-fetch: Compiler-enabled power-aware fetch throttling. *ACM Computer Architecture Letters*, 1, 2002.