Energy-Efficient Instruction Set Synthesis for Application-Specific Processors*

Jong-eun Lee[†]
jelee@poppy.snu.ac.kr

Kiyoung Choi[†]
kchoi@azalea.snu.ac.kr

Nikil D. Dutt[‡]
dutt@cecs.uci.edu

[†] EECS, Seoul National University, Seoul 151-742 KOREA
[‡] Center for Embedded Computer Systems, University of California, Irvine, CA 92697

ABSTRACT

Several techniques have been proposed to enhance the energy-efficiency of ASIPs (Application-Specific Instruction set Processors). While those techniques can reduce the energy consumption with a minimal change in the instruction set (IS), they fail to exploit the opportunity of designing the entire IS from the energy-efficiency perspective. In this paper, we present an energy-efficient IS synthesis technique that can comprehensively reduce the energy-delay product (EDP) of ASIPs through optimal instruction encoding, considering both the instruction bitwidth and the dynamic instruction count. Experimental results with a typical embedded RISC processor show that our technique can generate application-specific IS's that are up to 40% more energy-efficient over the native IS for several application benchmarks.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization General**]: Instruction set design (e.g., RISC, CISC, VLIW)

General Terms

Design, Algorithms

Keywords

Application-specific instruction set processor (ASIP), customization, instruction encoding, low power, energy-delay product

1. INTRODUCTION

It is well known that CISC IS's (Instruction Sets) are more energy-efficient than RISC IS's for the same microarchitecture [1]. However, it is not well known how we can utilize this observation to generate more energy-efficient IS's, especially when we are given the freedom to modify an IS on an application (or application domain) basis. With the recent development in soft IP's (Intellectual Properties) and configurable processors, IS customization has become possible and even necessary to make differentiation in today's competitive markets. Nonetheless, previous work on low-power ASIPs (Application-Specific Instruction Set Processors) has not been so ambitious as to fully exploit the flexibility of ASIPs and redesign the IS from the energy-efficiency perspective.

In this paper, we present an energy-efficient IS synthesis approach for application-specific processors. We optimize IS's under given microarchitectural constraints, as the design change in the datapath may incur significant engineering cost and thus not be desirable. With a fixed microarchitecture, specialization can be made in such areas as instruction encoding, the number of instructions, and the instruction bitwidth, all of which can be considered as instruction encoding in a broad sense. Thus, our objective is to find the best instruction encoding (manifested by RISC vs. CISC) that leads to the maximal energy-efficiency through fewer number of instructions fetched (reducing the instruction memory energy) or fewer number of execution cycles (reducing the processor core energy) or a balance of the two.

One of the critical elements of the proposed energy-efficient IS synthesis is reducing the instruction fetch energy through multiple dimensions of the code volume (the number of instructions fetched multiplied by the instruction bitwidth). While some previous low-power techniques can also have similar effects of reducing the code volume, only one dimension has typically been considered.[1] Our technique, on the contrary, addresses the multiple factors of the code volume and provides a comprehensive optimization framework for energy-efficient IS's. Also, our scheme generates a single IS as opposed to dual IS's (compressed, uncompressed) as the code compression techniques do; thus, it avoids the problems of dual IS's such as requiring an instruction decompressor (or re-map table) and, for some techniques, having to take care of the changes in the branch target addresses. It should be noted, however, that our IS customization framework assumes minor architectural changes in the datapath, allowed by the given architectural constraints, such as inserting additional muxes in front of functional units, as well as changing the instruction decoder logic. Our experimental re-

*This research was conducted while the first two authors were visiting UC Irvine, and supported in part by grants from NSF (CCR-0203813 and CCR-0205712) and Hitachi Ltd. We also thank members of the UCI EXPRESS compiler team for their assistance.

ISLPED'03, August 25–27, 2003, Seoul, Korea.

[1]For example, low-power instruction compression schemes [2][3] try to reduce the code volume by focusing only on the bitwidth of frequently occurring binary instruction patterns (thus not changing the number of instructions whether static or dynamic). Likewise, code size reduction techniques [4] aim to reduce the static code size, thus may not be effective for reducing the dynamic instruction count.

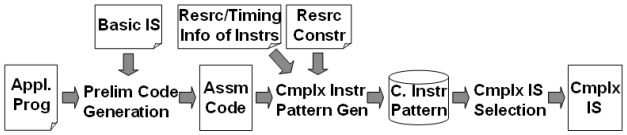


Figure 1: Instruction set synthesis flow.

sults show that our technique can generate application-specific IS's that outperform the native IS of a typical embedded RISC processor, not only in performance but also in energy and the EDP, up to about 40% with each metric.

The rest of the paper is organized as follows. In Section 2 we briefly discuss the previous work for energy-efficient ASIPs and in Section 3 we highlight the key elements of the encoding-oriented IS synthesis for ASIP customization. In Section 4 we derive the contribution of each instruction to the overall energy-efficiency based on an ASIP energy consumption model. We present our experimental results in Section 5 and conclude the paper.

2. PREVIOUS WORK

Previous work on low power techniques for ASIPs has mostly concentrated on bit pattern assignment of instructions, without changing the number of instructions in the IS. To reduce the switching activity and the dynamic energy consumption in IF (Instruction Fetch) registers of ASIPs, it was proposed to re-encode the opcode part of instructions so that the most frequent opcode sequences can have the smallest Hamming distances [5][6]. Or, exploiting the asymmetric energy consumption of some memory devices, even whole instructions can be re-encoded [7][8]. Also, for a more aggressive approach, removing unused or less useful instructions from the IS has been suggested [9].

While these techniques may fit well where only minimal changes can be made in the architecture, they are too conservative when a more aggressive IS redesign is preferred to seize the opportunity afforded by configurable processors. Contrastingly, we consider redesigning the IS from the energy-efficiency perspective to better exploit the flexibility of configurable processors. In [10], an IS synthesis technique is proposed considering instruction encoding, which is, however, developed for performance improvement and does not consider energy-efficiency. Our energy-efficient IS synthesis is based on this technique, but significantly extends it by considering various optimization goals and their efficacy.

3. SYNTHESIZING INSTRUCTION SETS

The encoding-oriented IS synthesis framework assumes *basic* instructions, which are provided by users once for each processor. A basic instruction is defined to have only one operation and is used to create new instructions for specific applications. These new application-specific instructions are called *complex* instructions because they are built by combining multiple basic instructions. Also, the basic instruction set (*basic IS*) provides a simple representation of the processor architecture, together with the resource constraints provided separately. Complex instructions are generated for each application or a set of applications representing a domain of applications.

Figure 1 illustrates the process of generating a complex IS. First the application and the basic IS are given as input. The application is compiled using a retargetable compiler targeted for the basic IS. This preliminary assembly code is used in the rest of the IS synthesis process. The actual synthesis process consists of two phases: complex instruction generation and instruction selection. In the

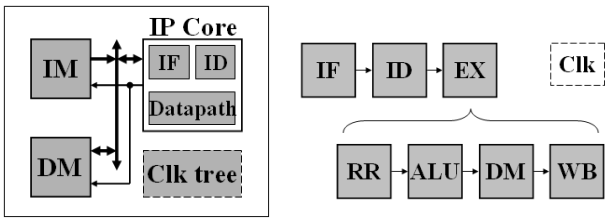


Figure 2: (a) The structural view and (b) the behavioral view of an example ASIP system.

complex instruction generation phase, a group of complex instructions are created for every sequence of up to N basic instructions appearing in the preliminary assembly code, where N is a design parameter. Then the most useful complex instructions are selected in the instruction selection phase.

Now, to select the most profitable ones to include in the final IS, we need to consider the benefit and the cost of choosing a complex instruction. If the objective is the performance, the benefit may be the number of cycles saved by using the complex instruction instead of the corresponding basic instruction sequence. The cost factor comes into play to take into account the instruction encoding constraint, i.e., there can be at most 2^{IW} (IW is the instruction bitwidth) number of distinct bit patterns that can be assigned to the selected instructions (including the basic instructions). Therefore, if a complex instruction i uses W_i bits to encode all its operands, 2^{W_i} number of bit patterns (called the *code space* used by i) need to be reserved for instruction i, which can be regarded as the cost of selecting i. Then, the instruction selection can be formulated as an optimization problem determining the set of complex instructions that maximizes the total benefit within the code space constraint (i.e., the code space of the selected ones should not exceed the total code space permitted).

However, this formulation involves two issues. First, the benefit and the cost of a complex instruction depend on what other complex instructions are already selected. Second, the effect (benefit) of a complex instruction depends on how the compiler will use the newly-generated complex instructions in the code generation phase as well as other compilation phases. [10] gives an efficient heuristic algorithm for the first issue, whereas for the second issue it assumes a certain deterministic procedure for the compiler to generate the code using the complex instructions, thus making it easy to estimate their benefit. This encoding-oriented IS synthesis framework provides a versatile IS synthesis environment where different CISC-like IS's can be generated for different optimization goals, in essence, by altering the definition of the benefit of a complex instruction.

4. ENERGY OPTIMIZATION

4.1 ASIP Energy Model

Figure 2 illustrates a simple ASIP chip including a processor IP core and memory blocks. From the behavior perspective, the same ASIP chip can be viewed, at the cycle level, as a pipeline of IF, ID, and EX stage operations. The CMOS dynamic energy of the whole ASIP can then be seen as $E_{IF} + E_{ID} + E_{EX}$, where E_{IF}, E_{ID}, and E_{EX} are the energy consumed by the operations in IF, ID, and EX stages, respectively. Also, the energy-delay product (EDP) of the

ASIP, for a given clock frequency, can be defined as

$$EDP = (E_{IF} + E_{ID} + E_{EX}) \cdot N_{cyc},$$

where N_{cyc} is the number of execution cycles of an application.

The energy consumed in each stage can be modeled, simplifying interrupts, pipeline flush, etc., as

$$E_{IF} + E_{ID} + E_{EX} = \\ [N_{ins} \cdot (e_{IM} + e_{IBus} + e_{IF})] + [N_{ins} \cdot e_{ID}] + \left[\sum_{op} N_{op} \cdot e_{op}\right],$$

where N_{ins} is the dynamic instruction count; e_{IM}, e_{IBus}, e_{IF}, and e_{ID} are the per-access energy consumption in the instruction memory, instruction bus, instruction fetch unit, and instruction decoder, respectively; and N_{op} and e_{op} are the number of operations and the per-operation energy consumption in the EX stage for each operation (group) op, respectively.

4.2 EDP Change Due to IS Customization

Let's consider the energy-efficiency (defined by EDP) of the basic IS (denoted by B) and the synthesized IS (denoted by C), which is B plus selected complex instructions. For the two IS's, let N^B_{cyc} and N^C_{cyc} be the numbers of execution cycles, and N^B_{ins} and N^C_{ins} be their dynamic instruction counts. Note that N^B_{cyc} and N^B_{ins} are fixed for a given application whereas N^C_{cyc} and N^C_{ins} are subject to optimization. From the IS synthesis procedure described in Section 3, we can assume the following relationships.

$$N^C_{cyc} = N^B_{cyc} - \sum_i R^{cyc}_i \cdot d_i \cdot \chi_i$$
$$N^C_{ins} = N^B_{ins} - \sum_i R^{ins}_i \cdot d_i \cdot \chi_i,$$

where R^{cyc}_i and R^{ins}_i are the expected reductions in the cycle count and the instruction count (respectively) if a complex instruction i is used once in place of a basic instruction sequence; d_i is the dynamic matching count, by which times the complex instruction i is used in the application[2]; and χ_i is a binary variable with the value 1 indicating i is selected.

Now, we make the following observations to derive the EDP difference. First, the EX stage operations such as ALU operations and memory operations are considered the same in terms of energy consumption for both B and C, assuming gated clock implementation to prevent unnecessary computation; hence, E_{EX} is the same for both IS's. Second, e_{IM}, e_{IBus}, and e_{IF} are considered the same for both B and C. Although the two IS's will have different binary patterns, resulting in different binary codes and different energy consumption, we assume that the per-access energy consumptions are the same for the two IS's.[3] Then, assuming that IF energy is much larger than ID energy increase (i.e., $e_{IM} + e_{IBus} + e_{IF} \gg e^C_{ID} - e^B_{ID}$), the EDP difference can be approximated as

$$\Delta EDP \approx \sum_i (a \cdot R^{cyc}_i + b \cdot R^{ins}_i) \cdot d_i \cdot \chi_i, \quad (1)$$

where constants a are b are defined as $a = E^B_{IF} + E^B_{ID} + E_{EX}$ and $b = E^B_{IF} + E^B_{ID}$.

[2]How many times a complex instruction will be used by the compiler depends on what the selected IS (C) is as well as the compiler's code generation algorithm. This dependency on C is taken care of by the selection heuristic by adjusting the benefit and cost values as complex instructions are selected.

[3]The instruction bitwidth is given as a parameter for the IS synthesis and thus is constant during the IS synthesis.

4.3 Modifying the Selection Algorithm

From the EDP difference of IS's in (1), we can quantify (approximately) each complex instruction's contribution to the reduction of EDP, or benefit Ben_i, as

$$Ben_i = a \cdot R^{cyc}_i + b \cdot R^{ins}_i. \quad (2)$$

Note that (2) can also represent the performance improvement and the energy reduction benefits by changing the constant values: $a = 1$ and $b = 0$ for performance improvement, and $a = 0$ and $b = 1$ for energy reduction.

To extend the IS synthesis framework for energy or EDP optimization, the benefit definition and the benefit updating part in [10] need to be changed according to (2). The newly introduced variable R^{ins}_i, which represents the instruction count reduction by using a complex instruction i instead of the corresponding basic instruction sequence, can be easily calculated—the number of basic instructions in the sequence minus one.

5. EXPERIMENTS

5.1 Experimental Setup

For our experiments, we used the MIPS microprocessor architecture [11], from which we defined a basic IS annotated with resource and timing information. For deriving complex instruction sets, we used a number of realistic benchmark applications covering multimedia (e.g., h.263 decoder, JPEG encoder), control-intensive (e.g., ADPCM coder/decoder), and cryptography (e.g., DES) domains. The benchmark applications were preprocessed using the EXPRESS retargetable compiler [12] targeting the basic IS to generate preliminary assembly code, which was used for the rest of the IS synthesis process.

While the MIPS architecture has 32-bit instructions, the native IS [11] of the MIPS uses the code space of only about 2^{30}, meaning that the native IS essentially uses only 30 bits, reserving the rest of the code space for future versions. Since the basic IS for the MIPS was defined with the code space of about 1.42×10^8, we used the code space of $2^{30} - 1.42 \times 10^8$ for the complex IS synthesis.

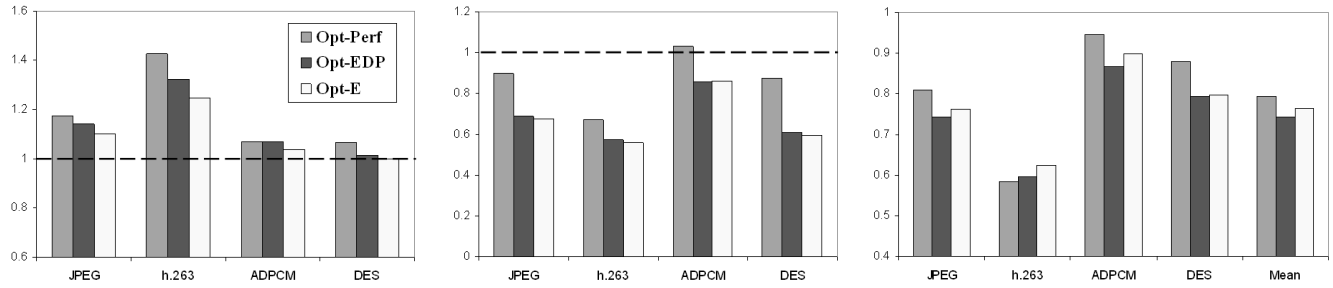
After the IS synthesis process, the application was recompiled using the same retargetable compiler targeted for the synthesized IS. The execution cycle counts and the instruction fetch counts were obtained through cycle-accurate simulation (at the basic block level) and profiling. We assume that the ID energy is the same for all IS's, as we compare the synthesized IS's with the native IS, which also has many complex instructions.

For the energy consumption estimation, we made the following assumptions on the architecture. We assumed that the instruction memory is large enough to fit each application so that there is no off-chip memory access for instruction fetch. To further simplify, we assumed that there is no instruction cache; thus, instruction memory energy consumption only depends on the number of instruction fetch. For the energy consumption ratio between IF, ID, and EX stages, we assumed that the ratio between $E^B_{IF} + E^B_{ID}$ and E_{EX} is 1:1 for all applications.[4] Note that this assumption also allows for an easy comparison of the EDP values for different IS's; that is,

$$R_{EDP} = [1 - 0.5(1 - R_{\#IF})] \cdot R_{\#CYC},$$

where R_{EDP}, $R_{\#IF}$, and $R_{\#CYC}$ are the ratios of the EDP, #IF, and the cycle count values (respectively) of two IS's.

[4]We base this ratio on the ARM920T power analysis result [13] taking into account different activation factors of components.



(a) Performance (b) #IF (Instruction Fetch) (c) EDP (Energy-Delay Product)

Figure 3: Performance, #IF, and EDP of the synthesized IS's, normalized to that of the native IS. The three bars per each application correspond to the three IS's synthesized with different optimization goals: Performance, EDP, and Energy (in order). The rightmost column in (c) is the geometric mean of the four benchmark results on EDP.

5.2 Improvement Through IS Synthesis

Recall that the strategy employed by our IS synthesis framework improves the native IS in two steps: 1) extract the basic IS from the native IS; and 2) build complex instructions on top of the basic IS. Therefore, the synthesized IS's always generated far better results compared to the basic IS in all the experiments performed, and these trivial results are not shown here. More importantly, in most cases the synthesized IS generated better results even compared to the native IS, as shown in Figure 3.

Figure 3 shows the results generated by the synthesized IS's, in terms of the performance (#cycles), #IF, and the EDP, normalized to that of the native IS. To see the effects of different optimization goals, i.e., performance, EDP, and energy (#IF), we synthesized three IS's for each benchmark application; thus, the three bars for each application represents the results of the three different synthesized IS's. The graphs show that the synthesized IS's can generate performance improvements of up to 43% or the IF reduction of up to 44%, though the improvements vary depending on the application as well as the optimization goal used. Also, when translated into EDP, the synthesized IS's can reduce the EDP by up to 42% (compared to the native IS), and 25% on average for all the applications using the EDP optimization. These results clearly show that the proposed technique can generate energy-efficient IS's for many applications in various domains.

From the figure, it is clear that optimizing for one metric does not necessarily lead to optimal results for other metrics as well, which confirms the need to consider the energy-efficiency metric more explicitly. Also, as expected, the best results for a metric were obtained by directly optimizing for the metric in most cases. There were minor exceptions, however: for the ADPCM benchmark the greatest energy reduction (the lowest #IF) was achieved by optimizing for EDP, and for the h.263 benchmark the greatest EDP reduction was achieved by optimizing for performance. This phenomenon is most likely due to the suboptimality of the instruction selection heuristic algorithm.

6. CONCLUSION

We have presented an energy-efficient IS synthesis technique that is based on an encoding-oriented IS synthesis framework. To comprehensively reduce the code volume, our technique optimizes the instruction encoding, considering both the instruction bitwidth and the dynamic instruction count. To apply the IS synthesis framework for energy-efficiency optimization, we formulated the EDP change due to IS customization and derived the contribution of each complex instruction. The experimental results show that our technique can generate application-specific IS's that outperform the native IS of a typical embedded RISC processor, in performance, energy, and the EDP, up to about 40% with each metric.

Our methodology to improve the IS of a processor from the energy-efficiency perspective focuses on the flexibility of ASIPs at the instruction level. In the future, we plan to investigate customization techniques at higher-levels (e.g., loops, functions), to exploit more opportunity available in future complex systems-on-chips.

7. REFERENCES

[1] J. Bunda et al. Energy-efficient instruction set architecture for CMOS microprocessors. In *Proc. Twenty-Eighth Hawaii Int'l Conf. System Sciences*, 1995.

[2] S. Chandar et al. Area and power reduction of embedded dsp systems using instruction compression and re-configurable encoding. In *Proc. ICCAD*, 2001.

[3] L. Benini et al. Selective instruction compression for memory energy reduction in embedded systems. In *Proc. ISLPED*, 1999.

[4] S. Liao, S. Devadas, and K. Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Trans. on Design Automation of Electronic Systems*, pages 12–38, 1999.

[5] S. Kim and J. Kim. Opcode encoding for low-power instruction fetch. *IEEE Electronics Letters*, 1999.

[6] L. Benini et al. Reducing power consumption of dedicated processors through instruction set encoding. In *Proc. Great Lakes Symposium on VLSI*, 1998.

[7] K. Inoue et al. Reducing power consumption of instruction ROMs by exploiting instruction frequency. In *Proc. Asia-Pacific Conf. Circuits and Systems*, 2002.

[8] T. Glokler and S. Bitterlich. Power efficient semi-automatic instruction encoding for application specific instruction set processors. In *Proc. Int'l Conf. Acoustics, Speech, and Signal Processing*, pages 1169–1172, 2001.

[9] W. Dougherty et al. Instruction subsetting: Trading power for programmability. In *Proc. Workshop on System Level Design*, 1998.

[10] J. Lee, K. Choi, and N. Dutt. Efficient instruction encoding for automatic instruction set design of configurable ASIPs. In *Proc. ICCAD*, pages 649–654, 2002.

[11] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, 2nd ed.* Morgan Kaufmann Publishers, 1997.

[12] University of California, Irvine. *EXPRESS Retargetable Compiler*. Project website http://www.cecs.uci.edu/~express.

[13] S. Segars. Low power design techniques for microprocessors. In *ISSCC*, 2001. (Presentation, also available at http://www.arm.com).