Battery Aware Instruction Generation for Embedded Processors

Newton Cheung†, Sri Parameswaran†§, Jörg Henkel‡
†School of Computer Science and Engineering, University of New South Wales, Australia
§National Information and Communications Technology Australia (NICTA), Australia
‡Department of Computer Science, University of Karlsruhe, Germany
{ncheung, sridevan}@cse.unsw.edu.au, henkel@informatik.uni-karlsruhe.de

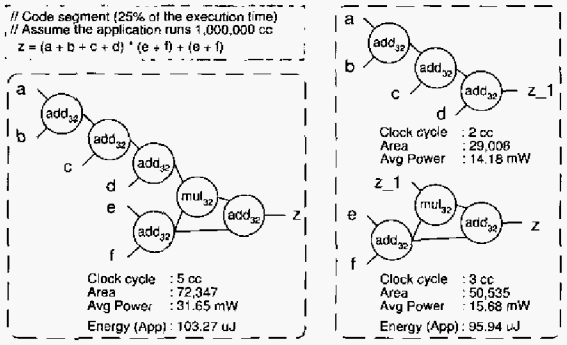
Abstract - Automatic instruction generation is an efficient method to satisfy growing performance and meet design constraints for application specific instruction-set processors. A typical approach for instruction generation is to combine a large group of primitive instructions into a single extensible instruction for maximizing speedups. However, this approach often leads to large power dissipation and discharge current, posing a challenge to battery-powered products. In this paper, we propose a battery-aware automatic tool to design extensible instructions which minimizes power dissipation distribution by separating an instruction into multiple instructions. We verify our automatic tool using 50 different code segments, and five large real-world applications. Our tool reduces energy consumption by a further 5.8% on average (up to 17.7%) compared to extensible instructions generated by previous approaches. For real-world applications, energy consumption is reduced by 6.6% on average (up to 16.53%) as well as an increase in performance for most cases. The automatic instruction generation tool is integrated into our application specific instruction-set processor tool suite.

1. INTRODUCTION

Application Specific Instruction-set Processors (ASIPs) are well-known for high performance and low power. ASIPs typically consist of a base processor core containing a base instruction set, and the capability to extend this instruction set through new extensible instructions. Extensible instructions are customized to replace computationally intensive code segments (groups of primitive instructions) in the application, satisfying performance and power dissipation constraints. Given an application and constraints, the design flow typically consists of four steps: 1) *identify* computationally intensive code segments of the application; 2) *generate* extensible instructions for identified code segments; 3) *select* extensible instructions based on the constraints; and 4) *evaluate* performance and constraints of the ASIP.

Previous approaches related to ASIPs have mainly focused on identifying large computationally intensive primitive instructions groups, and combining them into a *single* extensible instruction [7, 10, 12, 13, 24]. These approaches often maximize speedups and reduce execution time, and hence minimize energy consumption of the application. However, the drawback of these approaches is that power dissipation of extensible instructions is large and often consumes up to 20% of the total power dissipation when those instructions are executed. The variance of power dissipation distribution (between base processor and base processor plus extensible instructions) is not minimized. Thus, the variance of discharge current distribution is not minimized, which often leads to shortened battery lifetime [22].

Fig. 1 shows two designs that can be generated in order to replace a single code segment in the original software-based application. The code segment has six inputs, namely a, b, c, d, e, and f. The sum of first four inputs are multiplied and accumulated by the sum of last two inputs to produce an output, z, which executes for 25% of the time in the application. The base processor is a five-stage pipeline processor which runs at 222MHz with average power dissipation of 100mW. If an extensible instruction is to replace the code segment, then, by using previous state-of-the-art approaches [7, 10, 12, 13, 24], all the operations are combined into a single instruction, which is shown in Fig. 1a. The average power dissipation of the instruction is 31.65mW, which is consumed by different operations (i.e. five adders and multiplier) and registers used (i.e. seven registers). On the other hand, the designer can separate code segment into two instructions, as shown in Fig. 1b. The average power dissipation of these instructions are 14.18mW and 15.68mW respectively. The reduction of average power dissipation is because each of the instructions contains less operations and registers. Using the design in Fig. 1b, the energy consumption of the application[1] is reduced by 7% compared with the design in Fig. 1a. The reduction of energy consumption is because the instructions in Fig. 1b are executed sequentially and an extensible instruction only dissipates power when it is executed (see background). Therefore, energy consumption may be reduced by separating a large computationally intensive code segment into multiple instructions. *In the era of cost efficient, high performance and portability, it is critical to reduce power dissipation (to reduce energy consumption and extend battery lifetime) and explore as many of design points as possible. Therefore, an automatic tool for designing extensible instructions with battery-awareness is needed.*



(a) Single instruction (b) Multiple Instructions

Figure 1: Separates instruction to reduce energy

In this paper, we present an automatic instruction generation tool with battery-awareness which minimizes power dissipation of the instructions while maximizing speedups. We propose the separation of instructions into multiple instructions and utilize the slack of the instruction. As we will see, the automatic tool generates extensible instructions that reduce the energy consumption of the application by a further 6.6% (up to 16.53%) when compared with the instruction generated using previous approaches.

The rest of this paper is structured as follows. The next section describes the related work and emphasizes the contribution of our work. Section 3 gives the background. The problem statement is described in Section 4. The extensible instruction generation tool is presented in Section 5 and Section 6 discusses the experimental results. Finally, Section 7 concludes the paper.

2. RELATED WORK AND CONTRIBUTION

An overview of application specific instruction-set processors, their benefits and challenges are described in [15, 17]. In addition, commercial and research tool suites have shown that performance and power benefits can be orders of magnitude more efficient compared to general purpose processors when deployed in the same embedded systems [1, 2, 4, 5, 6].

Early research on instruction generation focuses on customizing instruction sets completely to satisfy design constraints. In [16], the authors describe an instruction set synthesis method for an application in a parameterized, pipelined micro-architecture. The work in [11] proposes an approach to generate multi-cycle complex instructions as well as single-cycle instructions for DSP applications. Due to the necessity of short design turnaround time, commercial vendors often provide a base processor to minimize enhancements necessary (i.e. extensible instructions) for designing a processor with given constraints. Thus, recent research has focused on instruction generation in reconfigurable processors and application specific instruction-set processors.

Instruction-set extension in reconfigurable processors combines application specific instruction-set processors with reconfigurable hardware (i.e. FPGA) which often formulates instruction-set extension as a compilation problem. The work in [13] proposes a performance-driven approach to generate instructions and allows operation duplication to enhance design space exploration. In [9], the authors describe an instruction synthesis method using resource sharing to minimize area efficiently. An instruction-set extension including local memory elements access is introduced in [8]. In [12], the authors describe a compiler approach to generate instructions for a VLIW architecture

[1] The energy consumption of an application is computed using the battery behavior model [20, 22], which is described in the background.

processor without constraining their size or shape. These approaches have mainly revolved around maximizing speedups of the application while minimizing area of the processor. None of these approaches focus on the energy consumption of the application.

Extensible instruction generation in application specific instruction set processors concentrates on generating instructions that satisfy the latency of the base processor while maximizing the performance and other constraints. In [7, 24], the authors describe methods to generate custom instructions via operation patterns matching. The work in [23] describes a scalable instruction synthesis that can be adapted by adding and removing operations. An automatic system to generate extensible instructions using different operation techniques is described in [14]. Although these approaches have shown energy reduction in their results, they are achieved by combining computationally intensive code segments into extensible instructions, and reduce execution time significantly with an incremental increase in power dissipation. In our automatic tool, we generate instructions to maximize speedups while minimizing the power dissipation in order to increase battery life for the application.

The novel contributions of our work are:

1. to separate instructions and utilize slack of the instruction to reduce the power dissipation of extensible instructions and to exploring fine-grain granularity in instruction generation;
2. for the first time, battery lifetime (battery behavior model) is taken into account for generating extensible instructions, and not just shortening the execution time which leads to energy reduction.

3. BACKGROUND

We use the battery behavior model described in [20, 22] to define the actual energy capacity that can be drawn from a battery. The battery lifetime, BL, can be defined as

$$BL = \frac{CAP}{P^{act}} \tag{1}$$

where CAP is the ideal energy capacity of a battery; and P^{act} is the actual power consumption of the circuit. The energy capacity of a battery is the amount of energy stored in a battery, which is measured in ampere-hours or watt-hours. Normally, the capacity of a battery decreases as the discharge current increases. In the analytical model, the actual current, I^{act}, that is taken out of the battery is

$$I^{act} = \frac{I}{\mu}, \quad 0 \leq \mu \leq 1 \tag{2}$$

where μ is the battery efficiency (or utilization) factor. The actual energy capacity, CAP^{act}, is

$$CAP^{act} = CAP \cdot \mu, \quad 0 \leq \mu \leq 1 \tag{3}$$

From Peukert's formula [20], an equation to evaluate the relationship between the battery capacity and the discharge current empirically is defined as

$$CAP = \frac{k}{I^{\alpha}} \tag{4}$$

where k is a constant determined by the chemical and physical design of the battery; I is the discharge current. For an ideal battery, α equals to 0, and for real battery, α ranges up to 0.7 for a typical load. Using eqn. 2 and eqn. 3, the battery efficiency factor can be defined in terms of the discharge current

$$\mu = f(I) = \frac{1}{I^{\alpha}} \tag{5}$$

where f is a monotonic-decreasing function [22]. In our case, we use α equals to 0.7. A comprehensive survey on battery modeling techniques is described in [18].

In [22], the authors showed that the battery efficiency is affected by the average discharge current as well as the average current profile. The actual power drawn out of the battery is defined as

$$P^{act} = V \int \frac{I}{\mu(I)} \cdot p(I) dI \tag{6}$$

where V is the voltage of the circuit; $\mu(I)$ is the battery efficiency factor; $p(I)$ is the probability density function of I. From eqn. 1 to eqn. 6, maximum battery lifetime is achieved when the variance of the discharge current distribution is minimized. If we assume that voltage is relatively constant during operations, then maximum battery lifetime is achieved when the variance of the power dissipation distribution is minimized.

As the design platform, we use the Xtensa processor from Tensilica Inc. [6]. It consists of a five-stage pipeline RISC base core with approx. 80 base instructions, plus the capacity to define specific functionality through extensible instructions, using Tensilica Instruction Extension (TIE), which coexist with the base instructions. An extensible instruction (in Xtensa) has hardware to clock-gate the instruction, such that the instruction can be turned on and off as needed. Therefore, the instruction is assumed to only dissipate power when it is executed.

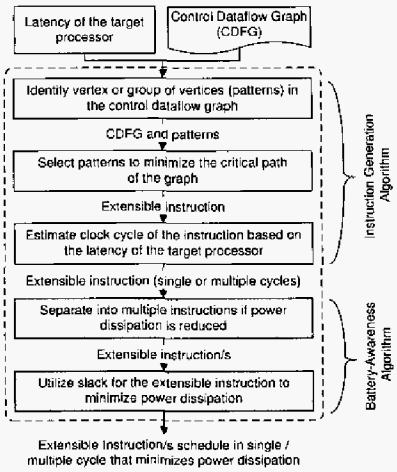


Figure 2: An overview of the automatic instruction generation tool

4. PROBLEM STATEMENTS

Our automatic tool is developed for the *generation* step in the design flow. The computationally intensive code segments have been identified in the *identification* step by using simulation and profiling [10].

Problem 1: Given a computational intensive primitive instruction group, generate extensible instruction(s) which maximize speedup of the extensible instruction(s) while the average power dissipation distribution of the extensible instruction(s) is minimized.

A Control Dataflow Graph (CDFG) $G(V, E)$ is a directed acyclic graph (DAG), where vertices, V, represent the primitive instructions from the base instruction-set of the target processor; edges, E, represent the data dependency between instructions. There are two properties of the graph: i) execution time[2], $et(G)$; and ii) average power dissipation, $apd(G)$. In addition, the latency (clock period) of the target processor is notated as $latency_{proc}$. Without loss of generality, we assume that $G(V, E)$ is a convex CDFG [7], and contains a maximum of ten input ports and ten output ports.

Using the preliminaries, we redefine problem 1 as *Problem 2:* Given a convex control dataflow graph (CDFG), $G(V, E)$, find subgraph(s) ($G' \subseteq G$) that cover all the vertices ($v \in V$) and minimizes the following constraints:

1. the execution time of the graph – the sum of the execution time of all subgraphs:
$$\sum_{G' \subseteq G} et(G')$$
2. the average power dissipation of the graph – the average of the subgraphs' average power dissipation:
$$\frac{\sum_{G' \subseteq G} apd(G')}{|G|}$$

Note that, minimizing execution time drives the design to make as few extensible instruction as possible because the execution overhead time (such as the register read/write time) may increase the overall delay. On the other hand, minimizing average power dissipation of the graph compels the design to be smaller multiple instructions. Due to the competing demands, this instruction generation problem is complex.

5. INSTRUCTION GENERATION

Our automatic instruction generation tool contains two algorithms (five phases): an instruction generation algorithm and battery-aware algorithm. An overview of the automatic instruction generation tool is shown in Fig. 2.

[2]Execution time is a time that requires to execute the instruction in the execution stage of the processor, which includes reading registers from the previous stage, executing operations, and writing registers.

5.1 Instruction Generation Algorithm

The goal of the instruction generation algorithm is to generate extensible instruction that maximizes the speedup and schedules the instruction into clock cycles based on the latency of the processor. The inputs of this algorithm are the latency of the target processor and a control dataflow graph that represents a computationally intensive code segment. The output is an extensible instruction. This algorithm consists of three phases: pattern identification, pattern selection, and clock cycle estimation.

Pattern identification is used to identify a vertex in the graph that can be replaced by well-defined operations[3], which could reduce the execution time of the graph. Additionally, if a group of vertices can be merged to a single pattern, then these are identified at this phase. For example, a number of sequential additions could potentially be integrated into a single adder. This phase begins the identification with each vertex, and expands to the connected vertices in order to find patterns until all the vertices are searched. All these patterns are passed on the next phase, along with the control dataflow graph.

Pattern Selection aims to select patterns which minimizes execution time of the control dataflow graph in order to maximize speedup. We propose a heuristic scheme to replace vertices with identified patterns only along the critical path. This phase first estimates the reduced execution time for each identified pattern along the critical path. The heuristic scheme then selects patterns that minimizes the critical path the most. The reason for searching the patterns only along the critical path is that the searching space is complex (possible number of patterns in the graph is 2^v). Thus, minimizing the critical path is sufficient to achieve maximum speedups in a short time, which reduces design turnaround time. However, after patterns are replaced, new critical paths may be formed. Therefore, we continue to minimize the critical path until the execution time of the graph is minimized.

Clock Cycle Estimation: After the critical path of the control dataflow graph is minimized, we estimate the number of clock cycles that the graph will take to execute when given the latency of the processor. The reason for estimating the clock cycle is to avoid violation of the processor's latency. If the graph violates the latency and does not schedule for multiple clock cycles, then the clock period of the base processor core must be increased - decreasing performance significantly. The clock cycle of the instruction is defined as

$$Clockcycle_{inst} = \left\lceil \frac{et(G)}{latency_{proc}} \right\rceil \quad (7)$$

where $et(G)$ is the execution time of the graph; and $latency_{proc}$ is the latency of the target processor.

5.2 Battery-Awareness Algorithm

The battery-awareness algorithm optimizes the battery lifetime of the product by minimizing power dissipation of extensible instructions. There are two phases: instruction separation and slack utilization.

Instruction Separation aims to separate the instruction into multiple instructions in order to reduce the power dissipation, extending the battery life of a product. The input is a single extensible instruction generated in the instruction generation algorithm, whereas the output is one or multiple extensible instructions. This phase searches every separation point[4] with multiple fan-ins and fan-outs, and constructs two subgraphs using separation point and evaluates the power dissipation reduction. The power dissipation evaluation is conducted using a power estimator that runs in a few seconds for each instruction. This phase then selects the separation point with the minimum power dissipation. These steps are iterated on the newly separated subgraphs in order to minimize the power dissipation until no further separation point is found. Fig. 3 shows an example of possible separation points. There are nine vertices (v_1, v_2, ..., v_9) in this control dataflow graph with ten inputs (a, b, ..., j) and two outputs (y, z). There are eight of the possible separation points ($C1$, $C2$, ..., $C8$) that separate the instruction differently. Assume all the vertices consume the same amount of power dissipation. The peak power dissipation of the instruction occurs before vertex, v_5. The predecessor vertex, v_4, must be scheduled in parallel with either of the vertices, v_1, v_2, or v_3, in order to maximize speedup. Peak power dissipation is doubled when vertices are overlapped, which often occurs when vertices have multiple fan-ins or fan-outs. However, separating instructions into multiple instructions may reduce the speedup, which in turn may lead to higher energy consumption. The *separation* function compares power dissipation and the number of clock cycles in the original instruction and the separated multiple instructions. The separation function is derived from the actual power equation (eqn. 6) and Amdahl's law [21] and is defined as

[3]It is referred to as patterns hereafter.

[4]A separation point is a cut that separates a graph into two subgraphs.

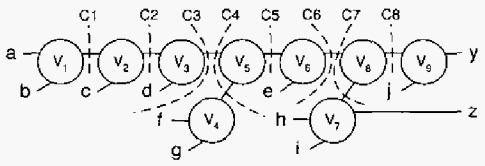
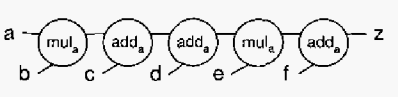


Figure 3: Separating instruction to reduce power



(a) A control dataflow graph

Patterns	Exe Time	Power
Add_a	3.3 ns	2.5 mW
Add_b	4.1 ns	2.2 mW
Mul_a	5.5 ns	10.5 mW
Mul_b	6.8 ns	9.8 mW

(b) Patterns

Figure 4: Utilizing the slack of the instruction

$$Separation_{Inst} = 1 - \sum_{x \in Inst} \eta_x \cdot \left(\frac{I_{proc} + I_x \theta_x}{I_{proc} + I_{orig\_inst}} \right)^{1+\alpha} \quad (8)$$

where x is the separated instructions from the original instruction, $Inst$; η_x is the percentage of probability density function[5] of instruction x compared with the probability density function of the original instruction; and θ_x is the percentage of current dissipation[6] reduction compared to the original instruction.

Slack Utilization aims to utilize the slack of the instruction. The clock cycle of the instruction is computed using eqn. 7, which is an approximated value. Therefore, this phase utilizes the slack of the clock cycle time of the graph to further minimize the power dissipation of the instruction. This phase begins with searching every path[7] of the graph including the non-critical path and ranks them, with the critical path as the highest. For each path, patterns that reduce the power dissipation while maintaining the clock cycle time are selected and replaced. Vertices are then marked and will not be replaced on the successive paths in order to maintain the clock cycle of the instruction. Fig. 4 shows an example for utilizing slack of the instruction. A control dataflow graph has six inputs, namely a, b, c, d, e, and f; and an output, z, which consists of three adders and two multipliers, as shown in Fig. 4a. Fig. 4b shows four patterns that can be used to replace vertices, and theirs delay and power dissipation. Assume the clock speed of the target processor is 222MHz. Using the instruction generation algorithm, the patterns add_a and mul_a are mapped to the addition and multiplication respectively to minimize the execution time. Therefore, the power dissipation and execution time is 28.5mW and 20.9ns respectively. Thus, the clock cycle required to execute this instruction is 4.64cc $\simeq$ 5cc. To utilize slack, we replace one of the additions with add_b to reduce the power dissipation to 28.2mW while maintaining the number of clock cycles in the instruction. This phase is efficient on non-critical paths of the graph. This approach further explores fine-grain granularity of instruction generation.

6. EXPERIMENTAL RESULTS

For evaluation purposes, we used the T1050.2 version of the Xtensa processor (0.18μ technology) from Tensilica Inc. [6], with a clock speed of 222MHz, consuming 100mW of power. All experiments were conducted on an Intel Xeon running at 2.4GHz (Quad), with 4Gb of RAM. Although these models are based on the Xtensa processor, the underlying method is general and can be applied to any platform of similar capability (ability to design extensible instructions). Fig. 5 shows the experimental platform for verifying our automatic instruction generation tool.

We conducted two separate experiments to determine the efficiency of the automatic instruction generation tool. We compare two different sets of instructions that are generated in the tool: i) instructions generated by the instruction generation algorithm (set 1); ii) instructions generated by the tool (with both algorithms) (set 2). In the first experiment, we examine the characteristics of the instructions such as area, power dissipation, execution time, and energy consumption of an application. Fifty code segments are selected with a wide range of instructions. These code segments are first applied in the automatic instruction generation tool. The generated instructions are then compiled and Verilog implementations are generated. We then synthesize these instructions to obtain area, power dissipation, and execution time using *Design Compiler* from Synopsys, Inc. [3] with a

[5]Probability density function is related to the number of clock cycles in the instruction.

[6]Current dissipation is related to the power dissipation where the voltage is assumed constant.

[7]A path is a group of vertices from an input port to an output port.

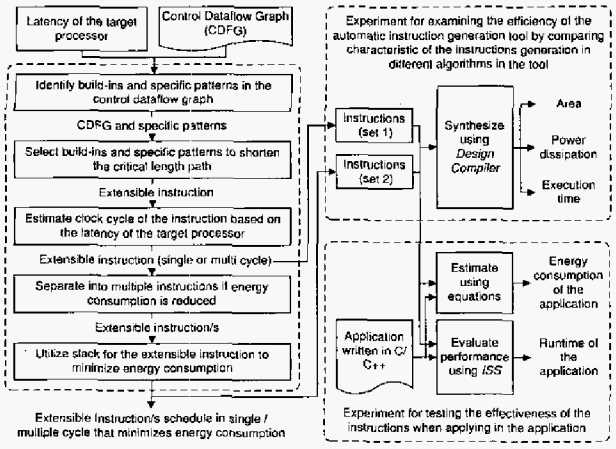


Figure 5: An experimental platform for verifying our automatic instruction generation tool

	Area [gates]		Power [mW]		Exe. Time [ns]		Energy [uJ]	
Inst Set	Ave.	Max.	Ave.	Max.	Ave.	Max.	Ave.	Max.
Set 1	15068	66348	21.98	61.51	10.52	26.97	76.48	123.01
Set 2	15357	63014	15.43	42.27	10.85	22.39	72.01	101.21
Red [%]	-1.92	5.03	29.80	31.28	-3.14	16.98	5.84	17.72

Table 1: The characteristics of the generated instructions (set 1 and set 2) for the fifty code segments

0.18μm cell library. The energy consumption is computed using the model described and the characteristics obtained. Each instruction is assumed to occupy 25% of the runtime in an application that runs 1,000,000 clock cycles.

The second set of experiments uses the identified code segments in five real-world applications: adpcm encoder (*adpcm*), g721 encoder (*g721e*), g721 decoder (*g721d*), epi encoder (*epie*), and epi decoder (*epid*) [19]. We apply these code segments and extract two different sets of generated instructions from the automated tool. We then replace the code segments with the instructions, and evaluate the runtime and the energy consumption of the application.

6.1 Evaluation Results

In our first experiment, we examine area, power dissipation, execution time, and energy consumption of the instructions. Table 1 shows the mean and maximum value of the characteristics (columns 2-9); and the characterisitics comparison between instructions in set 1 and instructions in set 2 (row 5). The area and execution time of the instructions in set 2 have increased 2% and 3% respectively when compared to the instructions in set 1. However, the instructions power dissipation in set 2 reduces 29.8% on average compared to the instructions in set 1. The reduction is achieved by using multiple instructions and reducing the number of registers used. In addition, the energy consumption of an application is further reduced by 5.8% on average (up to 17.7%) when compared to instructions in set 1. However, our tool is unable to reduce the energy consumption for 32% of the code segments in this experiment. The reason is that those code segments are too small to separation. If these instructions are separated, then the performance of these instructions are reduced significantly which leads to a large increase in energy consumption. Through experiments, our tool is more effective when the code segment is large and complex. Thus, separating and utilizing large computationally intensive code segments into multiple extensible instructions reduces the energy consumption of the application more effectively than combining a large code segment into a single instruction.

Table. 2 shows the characteristics of five real-world applications when different generated instructions are implemented and its original application (without any extensible instructions). The first column indicates the application name. The second column displays different extensible instructions used (see Fig. 5) that the application used. The third column shows the instruction average speedup while the fourth column shows the average power dissipation. The runtime and energy consumption of the application is shown in the next two columns. Finally, the last column displays the energy reduction comparison between the application using the instructions in set 1, and the application using the instructions in set 2. The average speedups of the instructions in set 1 and set 2 are within 5% of each other. The runtime of these applications using instructions in set 1 and set 2 are also within 5%. Note that, the runtime of the epi encoder, *epie*, is

	Inst Set	Instruction		Application		Energy Red.
		Average Speedup [x]	Average Power [mW]	Run Time [Sec]	Energy [mJ]	Comparison (Set 1 & Set 2) [%]
adpcm	Orig.	–	–	0.111	2.21	
	Set 1	2.2	15.27	0.095	2.04	
	Set 2	2.1	12.98	0.094	2.00	2.18
g721e	Orig.	–	–	1.58	31.53	
	Set 1	7.09	44.52	0.64	16.13	
	Set 2	7.22	28.45	0.61	14.12	12.43
g721d	Orig.	–	–	1.54	30.72	
	Set 1	7.09	44.52	0.65	16.33	
	Set 2	7.25	28.45	0.59	13.63	16.53
epie	Orig.	–	–	7.54	150.44	
	Set 1	18.52	34.67	0.48	11.86	
	Set 2	17.54	16.48	0.52	11.84	0.11
epid	Orig.	–	–	0.55	10.97	
	Set 1	8.15	34.67	0.16	3.94	
	Set 2	7.58	21.48	0.17	3.87	1.96

Table 2: The characteristics of the application when different instructions generated in the tool are applied

reduced significantly by 15.7x. The average power dissipation of the instructions in set 2 is 32.66% lower than the ones in set 1. Thus, the energy consumption of the application using set 2 is 6.6% (up to 16.53%) on average less than the energy consumption of the application using instructions in set 1. In addition, for the epi encoder, the energy reduction comparison is only 0.11%, which is due to an increase in the application's runtime.

7. CONCLUSIONS

We have presented an automatic tool for generating extensible instructions for ASIPs. As opposed to similar work of others we include a battery-aware algorithm that contains instruction separation and slack utilization. Using our tool, we are able to generate large and complex extensible instructions with low power dissipations. The automatic generation tool has been integrated into our processor tool suite. The results are as follows: the energy consumption of the real-world application which used the instructions generated in our tool is further reduced by up to 16.53% when compared to applications using instructions generated by previous methods. The automatic tool generates an extensible instruction in a few seconds for a given code segment.

8. REFERENCES

[1] Arctangent processor. ARC, Inc. (http://www.arc.com).
[2] Asip-meister. (http://www.eda-meister.org/asip-meister/).
[3] Design compiler. Synopsys, Inc. (http://www.synopsys.com).
[4] Jazz dsp. Improv Systems, Inc. (http://www.improvsys.com).
[5] Lisatek. CoWare, Inc. (http://www.coware.com).
[6] Xtensa processor. Tensilica, Inc. (http://www.tensilica.com).
[7] K. Atasu et al. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *DAC*, 2003.
[8] P. Biswas, V. Chodhary, K. Atasu, et al. Introduction of local memory elements in instruction set extensions. In *DAC*, 2004.
[9] P. Brisk, A. Japlan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *DAC*, 2004.
[10] N. Cheung, S. Parameswaran, and J. Henkel. Inside: Instruction selection/identification & design exploration for extensible processors. In *ICCAD*, 2003.
[11] H. Choi et al. Synthesis of application specific instructions for embedded dsp software. In *IEEE Trans. Computers*, 1999.
[12] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Micro*, 2003.
[13] J. Cong, Y. Fan, et al. Application-specific instruction generation for configurable processor architectures. In *FPGA*, 2004.
[14] D. Goudwin and D. Petkov. Automatically generating custom instruction set extensions. In *Compilers Architectrue and Synthesis for Embedded Systems*, 2003.
[15] J. Henkel. Closing the soc design gap. In *IEEE Computer Magazine, vol.36, Iss.9, pp119-121.*, 2003.
[16] I. Huang and A. Despain. Synthesis of application specific instruction sets. In *IEEE Tran. on Computer Aid Design*, 1995.
[17] K. Keutzer, S. Malik, and A. R. Newton. From asic to asip: The next design discontinuity. In *ICCD*, 2002.
[18] K. Lahiri et al. Battery-driven system design: A new frontier in low power design. In *ASP-DAC/VLSI*, 2002.
[19] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communica-tons systems. In *Int. Symp. on Microarchitecture*, 1997.
[20] H. D. Linden. *Handbook of Batteries*. McGraw-Hill, 1995.
[21] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantivtative Approach*. Morgan Kaufmann Publishers, 1989.
[22] M. Pedram and Q. Wu. Battery-powered digital cmos design. In *IEEE Tran on VLSI*, 2002.
[23] F. Sun, A. Raghunathan, S. Ravi, and N. K. Jha. A scalable application specific processor synthesis methodology. In *ICCAD*, 2003.
[24] F. Sun, S. Ravi, A. Raghunathan, and N. Jha. Custom-instruction synthesis for extensible-processor platforms. In *IEEE Tran. on Computer Aid Design*, 2004.