Energy Efficient Instruction-set Extension Considering Inline Expansion

Sho NINOMIYA, Keishi SAKANUSHI,
Yoshinori TAKEUCHI, and Masaharu IMAI

Graduate School of Information Science and Technology Osaka University
1-5 Yamadaoka, Suita, Osaka, 565-0871 JAPAN
{s-ninomy, sakanusi, takeuchi, imai}@ist.osaka-u.ac.jp

Abstract— To reduce energy consumption of applications in embedded systems, instruction-set extension suitable for the application is necessary on ASIP. Inline expansion, one of the software optimization, is not considered in conventional instruction set extension method. In this paper, we propose energy efficient instruction-set extension method considering inline expansion. The experiment shows the proposed method reduce more energy consumption.

I. Introduction

Embedded systems such as mobile phones and medical instruments are required high performance, compact, and low power. To meet these requirements, a configurable processor like ASIP (Application Specific Instruction-set Processor), which has high programmability and high performance, is widely used in embedded systems. ASIP can enhance application's performance by adding custom instructions. Recently, compact battery powered portable embedded systems become popular, so reducing application's energy becomes great concern.

For energy-efficient applications on embedded systems, both hardware and software approaches to reduce application's energy are considered. One of the hardware approaches is instruction set extension. Instruction set extension aims for reducing execution cycles by replacing an instruction sequence with a custom instruction under the functional unit constraints [1]. One of the software approaches is inline expansion. Inline expansion replaces a function call-site with a body of callee function. In this approach, code size of the application will increase owing to duplication of callee functions, but execution cycles will decrease because of overhead eliminations in function calling. In [2], deciding the set of call-sites to be inlined with some metrics, and reducing execution cycles accompanied with overhead elimination in function calling.

Since both hardware and software approaches are independent, it is able to minimize execution cycles by each optimization. However, hardware approaches do not take account of software approach and vice versa. In [1], customized instruction set that minimizing execution cycles is determined considering the instruction reduction by the custom instruction and the number of repetition. Even if the customized instruction set which does not reduce execution cycles most, it can makes code size smaller, and the more call-site can be inlined. After all, execution cycles of the application will be reduced.

In this paper, we propose energy-efficient instruction set selection approach for ASIP considering inline expansion.

The rest of the paper is organized as follows. In section 2 and 3, we explain about instruction set extension and inline expansion, respectively. In section 4, we discuss relations between instruction set extension and inline expansion. In section 5, we formulate an instruction set selection problem. In section 6, we propose energy-efficient instruction set deciding algorithm. We present our experimental results in section 7 and conclude the paper.

II. Instruction-Set Extension

In instruction set extension, designers add new custom instructions to the original instruction set in order to improve application's performance. Generally, instruction set extension is divided into two stages, instruction generation and instruction selection. First, the instruction sequences which executed frequently are extracted, and a set of them is selected in order to minimizing execution cycles of the application.

A. Custom Instruction Generation

In custom instruction generation, control flow graphs are constructed, and subgraphs corresponding to custom instructions are identified from them. Generally, this subgraph identification takes a lot of time because control flow graph has large amount of subgraph candidates.

In [3], to shorten instruction generation time, custom instructions are obtained by integer linear programming based algorithm. This algorithm takes account of the hardware constraints such as area of processor and inputs and outputs of functional units.

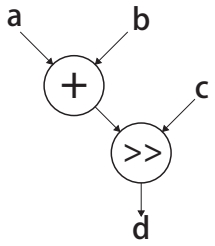

Fig. 1. DFG for identifying the custom instruction

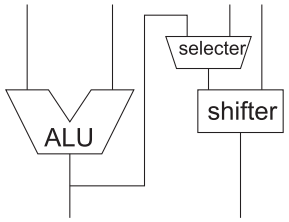

Fig. 2. Datapath constructed according to DFG of Fig.1

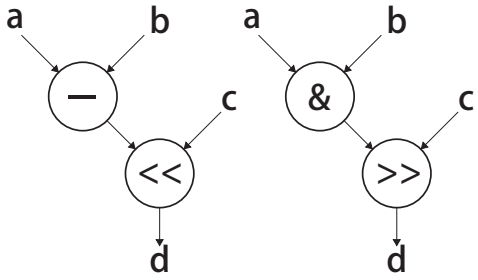

Fig. 3. Example of DFGs that datapath of Fig.2 can operate

In [4], the authors proposed template matching method for short-time identification.

B. Custom Instruction Selection

In custom instruction selection, a set of custom instructions is determined from custom instruction candidates. This combination has to meet the constraints such as area of processor and number of custom instructions. In [5], instruction selection is performed with greedy approach which evaluates the subgraph one by one and selects the subgraphs in descending order.

C. Energy Reduction by instruction set extension

After instruction set extension, performance such as area, delay, energy consumption, etc. will change due to the alternation of datapaths. When identifying the custom instruction from the subgraph (Fig.1), this arithmetic 'right shift' after 'add' operation is processed through shifter to ALU, and the datapath is constructed according to Fig.2. In this alternation, the functional units, selectors and signal wires connecting them will be added to the processor. In case ALU and shifter operations are performed in one cycle, performance of the processor will increase as adding custom instructions to the processor's instruction set if ALU-shifter operation like 'left shift' after 'or' and 'right shift' after 'add' mentioned above is performed in one cycle.

Especially, in the embedded system, the number of functional units which can be added to the processor is limited due to the area constraints, so custom instructions are added under the fixed functional units. It is more important that which custom instructions should be added, in the case of instruction set extension with no additional functional units. For instance, the subgraphs in Fig.3 are considered as candidates, in datapath of Fig.2. In instruction set extension on the architecture of the fixed functional units, it is able to assume that the processor consumes almost constant power even if new instructions are added, so the execution cycle reduction will directly affect the energy consumption reduction.

III. Inline Expansion

Inline expansion replaces a function call-site with a body of callee. Inline expansion reduces execution cycles accompanied with an overhead elimination. On the other hand, the code size of application will increase because callee functions are duplicated in caller functions. Therefore, the tradeoff between code size and execution cycles is considered when inline expansion is performed.

A. Analysis for Inline Expansion

Information such as the code size of functions and the function call count is needed for inline expansion. This information is obtained by profiling the target application. Generally, the relationship between caller functions and callee functions is described with a callgraph, which expresses a function and a call-site as a node and an edge, respectively. Figure 5 shows a callgraph structured with the functions and call-sites in Fig.4. Especially, dynamic profiling can obtain function call count when application is running. Dynamic profiling is essential for inline expansion because the more overhead is eliminated if functions called frequently are inlined.

B. Priority of Function Inlining

Figure 6 shows function and call-site after inlining funcA and funcB, and Fig.7 shows callgraph of Fig.6. The code size and the function call count of each function differ.

A problem to maximize reduction of execution cycles by inline expansion is equivalent to knapsack problem which

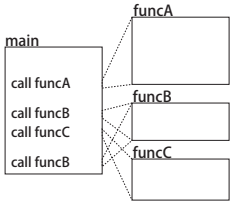

Fig. 4. Functions and call-sites

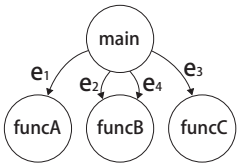

Fig. 5. Callgraph of Fig.4

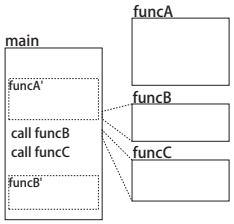

Fig. 6. Functions and call-sites after inlining

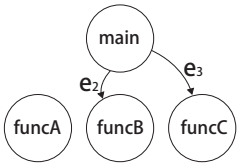

Fig. 7. Callgraph of Fig.6

maximizes the weight of edges in the callgraph. It is well known that knapsack problem is NP-complete [6]. In [2], the authors proposed heuristic approach which minimizes execution cycles of the application, with the metrics for deciding priority of function inlining.

C. Energy Reduction by Function Inlining

After calculation of energy consumption under specific architecture and instruction set of the processor, average power per one cycle can be evaluated. The energy reduction by function inlining is the product of execution cycles reduction by function inlining and average power per one cycle of the processor.

IV. instruction set extension and Inline Expansion

Since instruction set extension is a hardware approach and inline expansion is a software approach, and generally, they are performed individually. However, these optimizations are not completely independent. In this section, we explain the effect on inline expansion by instruction set extension.

Figure 8 shows an allocation of inlined application on the memory in different instruction sets. In IS_C (instruction set including custom instructions which generated from funcC), code size of funcC and inlined call-site e_4 will be reduced compared to that in the original instruction set. On the other hand, in IS_A (instruction set including custom instruction which generated from the funcA),

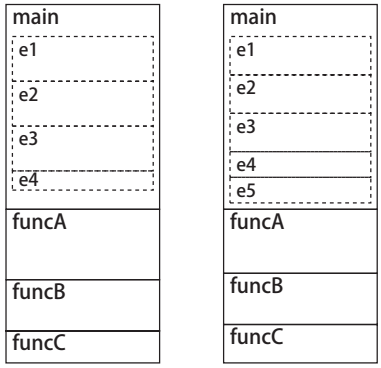

Fig. 8. Allocation of inlined application on the memory in different instruction

code size of funcA and inlined call-site e_1, e_2 and e_3 will be reduced, and additional call-site e_5 will be inlined.

In IS_C, let energy reduction by custom instruction be W_C and energy reduction by inline expansion be W_{IE_C}. In IS_A, let energy reduction by custom instruction be W_A and energy reduction by inline expansion be W_{IE_A}. When $W_C + W_{IE_C} < W_A + W_{IE_A}$, the application energy in IS_A will be much reduced even in case of $W_C > W_A$. Consequently, instruction set extension and inline expansion have to be considered simultaneously, in order to minimizing energy consumption of the application.

V. Energy-efficient Instruction Set Deciding Problem

A. Problem Statements

In this section, we explain inputs, outputs, constraints, and object of the energy-efficient instruction set deciding problem.

Input Inputs of the problem are the basic instruction set of the processor, the custom instruction candidates and the instruction sequence of application.

Output Output of the problem is a customized instruction set which satisfying given constraints.

Constraint Constraints of the problem are area, maximum number of extendable custom instructions and memory size limitation.

Object Object of the problem is minimizing energy consumption of the application considering execution cycle reduction by inline expansion.

VI. Instruction set extension method considering Inline Expansion

In this section, we propose energy-efficient instruction set extension method considering execution cycle reduction by inline expansion. In the proposed method, optimal instruction set is obtained with heuristic by evaluating energy consumption when the customized instruction set is implemented.

A. Energy-efficient Instruction Set Deciding Algorithm

Algorithm to obtain optimal instruction set considering inline expansion is as follows.

- Calculate inlining priority of call-sites by dynamic profiling.
- Store reduction of execution cycles and code size to look-up table for each call-site e_i and custom instruction candidates CI_i.
- Repeat the following process under the constraints of area and maximum number of custom instructions.
  1. Calculate energy consumption of the target application for each custom instruction candidates CI_i.
  2. Add the custom instruction minimizing application energy to the solution.

First, obtaining the inlining priority of call-sites with evaluating function (in Sec.B.) from the dynamic profiling information of the application.

Next, calculating reduction of execution cycles and code size for each call-site e_i when the custom instruction CI_i to be implemented, and store them to the look-up table (in Sec.C.).

After that calculating the application's energy for each custom instruction with the inlining priority and look-up table.

Finally, custom instruction which minimizes application's energy is added to the solution.

B. Inlining Priority of Call-sites

We explain about the evaluating function to obtain the inlining priority and variables used in this function, proposed in [2].

Control Flow Graph Control flow graph expresses relations between basic blocks. Nodes are corresponding to basic blocks and edges are corresponding to branches to other blocks in this graph.

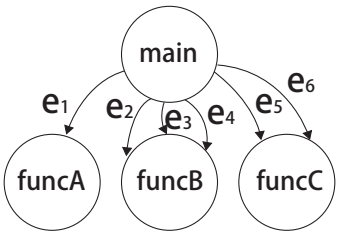

Fig. 9. An example of Functions and call-sites

Critical Path Length The length of the critical path of the procedure in terms of basic blocks. This estimation is performed in the front-end phase using the control flow graph. At every branch point in the control flow graph, an edge with the highest profile count among the outgoing edges is chosen as part of the critical path. All back edges are ignored.

Level Leaves of a call-graph is defined as at level 1. The level of a node is computed as 1 greater than the maximum level of its children.

Normalized Level This element is maintained as part of the edge summary since inlining of that particular edge causes the level of the caller to be set to the normalized value.

Evaluating Function The evaluating function described in Fig.1 shows how suitable for inlining a call-site is, where N_s^t is code size of the caller, N_s^s is code size of the callee, N_p^t is critical path length of the caller, N_p^s is critical path length of the callee, E_{nl} is normalized level, E_c is execution count of the function call, E_s is instruction count of an overhead in function calling, E_o is eliminated instruction count by inline expansion, and E_{ic} is instruction cache pressure which represents the execution count increase due to cache misses.

$$\frac{E_c \times E_s \times E_o}{(N_s^t + N_s^s) \times E_{nl} \times N_p^t \times N_p^s \times E_{ic}} \quad (1)$$

In the proposed method, the evaluating function (eq.(1)) determines the inline priority of call-sites. For example, for callgraph in Fig.9, four call-sites (e_1, e_2, e_4, e_5) are inlined in order (Fig.10) under the memory limitation. We define $t(e)$ as a binary variable which takes 1 when a call-site e is inlined or 0 when a call-site e is not inlined. Thus, $t(e_1), t(e_2), t(e_4), t(e_5)$ takes 1 and $t(e_3), t(e_6)$ takes 2.

C. Calculation of Execution Cycles and Code Size

In the proposed method, reduction of execution cycles and code size are estimated when several custom instruc-

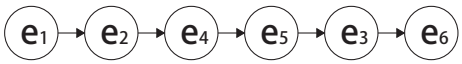

Fig. 10. Inlining Priority of call-sites in Fig.9

TABLE I
AN EXAMPLE OF CODE SIZE REDUCTION LOOK-UP TABLE

	F_1	F_2	F_3	$\cdots$	F_n
I_1	80	120	116	$\cdots$	70
I_2	76	120	110	$\cdots$	66
I_3	80	112	110	$\cdots$	70
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
I_m	78	118	110	$\cdots$	66

tions to be implemented. However, this estimation needs high computation for every combination of custom instructions. Therefore, in the proposed method, reduction of execution cycles and code size are calculated, and stored to the look-up table.

D. Calculation of Energy Consumption

Equation (2) shows energy consumption of the target application after function inlining and custom instruction I implemented, regarding $W_{ie}(I)$ as energy reduction after inline expansion and $W_{ise}(I)$ as energy consumption of the application after instruction set extension.

$$W(I) = W_{ise}(I) - W_{ie}(I) \tag{2}$$

$W_{ise}(I)$ is calculated as eq.(3), where E_p denotes average power per a cycle , C_a denotes execution cycles of the application, $C_{ise}(I)$ denotes increase of execution cycles after instruction set extension and $E_{ise}(I)$ denotes increase of average power per a cycle after instruction set extension.

$$W_{ise}(I) = (E_p + E_{ise}(I)) \times (C_a - C_{ise}(I)) \tag{3}$$

If hardware resources are fixed, selectors and signal wires increase slight area, so the increase of average power per a cycle is regarded as almost zero. Moreover, $W_{ise}(I)$ in Fig.3 is regarded as $W_{ise}(I)$ in Fig.4.

$$W_{ise}(I) = E_p \times (C_a - C_{ise}(I)) \tag{4}$$

$W_{ie}(I)$ is calculated as eq.(5), with the variable $t(e)$ defined already in section B..

$$W_{ie}(I) \quad = \quad \sum_e E_p \times C_{ie}(I, e) \times t(e) \tag{5}$$

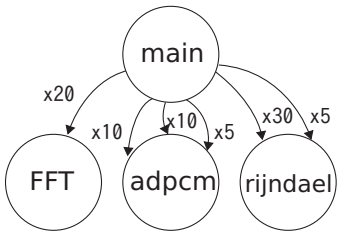

Fig. 11. An example of callgraph used in this experiment.

Consequently, energy consumption of the application considering inline expansion is computed as the eq.(2).

VII. EXPERIMENTS

In this section, we explain about the experimental setup and it's results.

A. Experimental Setup

We customized a processor, BrownieSTD32, which has 32 bit RISC architecture. In this experiment, the customized BrownieSTD32 has 2Kbytes memory, two ALUs and a barrel shifter, and we assume that the number of custom instructions is up to 6.

We executed 20 applications which have lots of call-sites and functions. These functions perform various processings such digital signal processing as (FFT, adpcm and CRC32) and cryptography (rijndael and sha) [7]. An example is shown in Fig.11. We optimized the target applications by the following five ways.

No optimization applications without inline expansion on the original instruction set.

Inline applications with inline expansion on the original instruction set.

ISE applications without inline expansion on the customized instruction set.

Sequential applications with inline expansion on the customized instruction set generated by [1].

Proposed applications with inline expansion on the instruction set generated by the proposed method.

The applications are compiled with GNU C Compiler (gcc), and the optimization option is '-O2', which performs almost all the optimizations except them having a tradeoff between code size and performance.

B. Experimental Results

We explain results of this experiment to show effectiveness of the proposed method.

TABLE II
ENERGY CONSUMPTION BY VARIOUS OPTIMIZATIONS.[μJ]

Application	Method	Energy reduction by inline expansion	Energy reduction by custom inst.	Sum of energy reduction	Whole energy consumption
Min	No optimization	0.00	0.00	0.00	253.03
	ISE	0.00	17.21	17.21	235.82
	Inline	21.32	0.00	21.32	231.71
	Sequential	24.53	17.21	41.74	211.29
	Proposed	24.53	17.21	41.74	211.29
Max	No optimization	0.00	0.00	0.00	278.11
	ISE	0.00	14.48	14.48	263.63
	Inline	24.09	0.00	24.09	254.02
	Sequential	24.36	14.48	38.84	239.27
	Proposed	28.90	13.02	41.92	236.19

Table II shows energy reduction and whole energy consumption of applications by No optimization, Inline, ISE, Sequential, and Proposed.

In the table, Max indicates the application which has maximum energy difference between Sequential optimization and Proposed optimization. Proposed optimization reduces more energy than No optimization by 15%. On the other hand, Sequential optimization reduces more energy than No optimization by 14%. Therefore, the proposed method is performed effectively on this application.

In the table, Min indicates the application which has minimum energy difference between Sequential and Proposed. Proposed reduces more energy than No optimization by 16%, and Sequential reduces same energy. This is because the instruction set generated by Proposed optimization is same as that by Sequential optimization.

On average of 20 applications, Proposed reduced more energy than No optimization by 15.7%, and Sequential reduced more energy than No optimization by 15.2%.

These results show that the proposed method is much better way for energy-efficient application.

VIII. CONCLUSION

We proposed energy-efficient instruction set extension method considering inline expansion. The proposed method reduces more energy than optimizing individually for the application using a lot of call-sites. Future work is experiments with practical application and instruction set extension method considering other software optimizations.

REFERENCES

[1] N. Cheung, S. Parameswaran, and J. Henkel, "Inside: Instruction selection/identification design exploration for extensible processors," in *Computer Aided Design, 2003. ICCAD-2003. International Conference on*, pp. 291 – 297, nov. 2003.

[2] D. Chakrabarti and S.-M. Liu, "Inline analysis: beyond selection heuristics," in *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pp. 291 – 297, march 2006.

[3] C. Galuzzi, E. M. Panainte, Y. Yankova, K. Bertels, and S. Vassiliadis, "Automatic selection of application-specific instruction-set extensions," in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '06. ACM, pp. 160–165, 2006.

[4] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Synthesis of custom processors based on extensible platforms," in *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '02. ACM, pp. 641–648, 2002.

[5] K. Seto and M. Fujita, "Custom instruction generation with high-level synthesis," in *Application Specific Processors, 2008. SASP 2008. Symposium on*, pp. 14 –19, june 2008.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[7] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. IEEE Computer Society, pp. 3–14, 2001.