Power Analysis and Low-Power Scheduling Techniques for Embedded DSP Software

Mike Tien-Chien Lee, Vivek Tiwari*, Sharad Malik*, and Masahiro Fujita

Fujitsu Laboratories of America, Inc.
77 Rio Robles,San Jose, CA 9513
{lee,fujita}@fla.fujitsu.com

*Department of Electrical Engineering
Princeton University, Princeton, NJ 08544
{vivek,malik}@ee.princeton.edu

Abstract

This paper describes the application of a measurement based power analysis technique for an embedded DSP processor. An instruction-level power model for the processor has been developed using this technique. Significant points of difference have been observed between this model and the ones developed earlier for some general-purpose commercial microprocessors [1, 2]. In particular, the effect of circuit state on the power cost of an instruction stream is more marked in the case of this DSP processor. In addition, the DSP processor has a special architectural feature that allows instructions to be packed into pairs. The energy reduction possible through the use of this feature is studied. The on-chip Booth multiplier on the processor is a major source of energy consumption for DSP programs. A micro-architectural power model for the multiplier is developed and analyzed for further energy minimization. A scheduling algorithm incorporating these new techniques is proposed to reduce the energy consumed by DSP software. Energy reductions varying from 11% to 56% have been observed for several example programs. These energy savings are real and have been verified through physical measurement.

1 Introduction

Embedded computing systems are characterized by the presence of application specific software running on specialized processors. These processors may be off the shelf digital signal processors (DSPs) or application specific instruction-set processors (ASIPs). A large fraction of these applications are power critical. However, there is very little available in the form of design tools to help embedded system designers evaluate their designs in terms of the power metric.

Recently an instruction-level power model was developed for two general-purpose commercial microprocessors [1, 2], which is based on the *base cost* and the *overhead cost* of an instruction, obtained by physical current measurements. The base cost of a given instruction is defined as the average current drawn by the processor during the repeated execution of the instruction. The "overhead cost" was needed to account for the effect of circuit state change for an instruction sequence consisting of different instructions. However, the impact of this effect on the overall power cost of programs was found to be limited for these large general-purpose microprocessors. The aim of this study is to analyze this and other issues related to software power consumption, in the context of a smaller, more specialized processor.

A Fujitsu embedded DSP processor, referred to as the target processor from here on, is used for our study. This processor is used in several Fujitsu embedded applications and is representative of a large class of DSP processors. The analysis results are used in this paper to develop an instruction-level power model that makes it possible to evaluate the power cost of programs that run on the target DSP processor. It is observed that the effect of circuit state is more marked for this processor. This suggests that changing the instruction order by an appropriate scheduling of instructions can lead to a reduction in the power cost of a program. This issue has been explored to a limited extent in earlier works [3, 4]. In [3], the study shows that faster programs consume less energy. So optimizing software performance through instruction scheduling can minimize the energy consumption. In [4], only the power consumed by the controller of a processor is targeted for minimization. The power cost of different instruction schedules is estimated by counting transitions on an RTL level model of the control-path. As is well recognized by now, such an estimation method is only a rough measure of the actual power cost. Furthermore, the increase in energy cost due to longer schedules is not considered. The scheduling approach that we propose overcomes these limitations since it is based on actual energy costs obtained through physical measurements.

The DSP processor has a special architectural feature that allows instructions to be packed into pairs. The energy reduction possible through the use of this feature is studied as well. Furthermore, since the on-chip Booth multiplier is a major source of energy consumption for DSP programs, a micro-architectural power model for the on-chip Booth multiplier is developed and analyzed for further power minimizations. Based on this microarchitecture model, an effective technique of local code modification by operand swapping is proposed to further reduce power consumption. A low-power code scheduling method is then presented to automatically apply the above techniques to any given piece of code. Experimental results on several example DSP programs show energy reductions ranging from 11% to 56%. The energy savings have been verified through physical measurement. It should also be noted that the energy reduction essentially comes for free. It is obtained through software modification, and thus, entails no hardware overhead. In addition, there is no loss of performance since the running times of the modified programs either improve or remains unchanged.

2 Target DSP Processor Architecture

The target embedded DSP processor used for our study is a Fujitsu 3.3V, $0.5um$, 40MHz CMOS processor. Special architectural features relevant to the rest of the paper are: (1) 4 24-bit data registers (A, B, C, D), (2) a fast MAC (Multiply-and-ACcumulate) unit using a Booth multiplier, (3) latched operands for the Booth multiplier to reduce unnecessary switching, (4) two on-chip RAM banks for simultaneously reading into two registers, and (5) instruction packing.

3 Current Measurement

The applications of the target DSP processor run on the limited energy available in a battery. Thus, energy consumption is the focus of attention. Since Vdd (supply voltage) and τ (the cycle period) are known and fixed, energy ($E = (I * Vdd) * (n * \tau)$) is proportional to the product of I (average current) and n (number of cycles). Given n for a program, we just need to measure the average current, I, to calculate E. The product, $I * n$, is the measure used to compare the energy cost of programs in this paper.

The target processor is part of a personal computer evaluation board. It can be programmed through a monitor program running on a personal computer. Using the monitor, the DSP instructions can be downloaded to the off-chip instruction memory, while the input data can be stored in the on-chip RAM of the DSP processor. The current drawn by the DSP processor is measured through a standard off-the-shell, dual-slope integrating digital ammeter, in the same way as discussed in detail in [1].

4 Instruction Packing for Low Power

A special architectural feature of the target DSP processor is the capability of packing an ALU-type instruction and a data transfer instruction into a single instruction codeword for simultaneous execution. This feature is called *instruction packing*.

We found that using the packed instruction always leads to a reduction in energy. The reason for this is that the average current for the packed instruction is only slightly more than the average current for the sequence of the two unpacked instructions. Thus, the reduction of one execution cycle, more than offsets the slight current increase, leading to a large overall energy reduction.

This is graphically illustrated in Figure 1 for a certain set of operands. The area under the solid and dotted graphs is proportional to the energy consumption for the packed and unpacked instructions, respectively. As can be seen, the average current drawn for the packed instruction is only marginally higher than for the unpacked instructions. However, since the unpacked instructions complete in twice the number of cycles as the packed instructions, the total energy consumed by the unpacked instructions is much larger, about twice as much as the packed instructions.

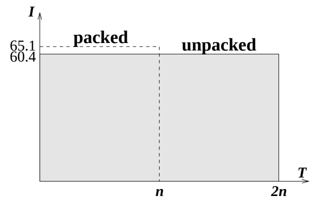

Figure 1: Comparison of energy consumed by packed and unpacked instructions.

Table 1: An example of a sequence four instructions where the overhead cost between 1 and 3 cannot be ignored.

number	instruction	base	cycles
1	`MUL:LAB (X0+1),(X1+1)`	37.2	1
2	`NOP`	14.4	1
3	`MUL:LAB (X0+1),(X1+1)`	36.6	1
4	`NOP`	14.4	1

The explanation for the above observations may lie in the fact that there is a certain underlying current cost associated with the execution of any packed or unpacked instruction, which is independent of the functionality of the instruction. This is the cost associated with fetching the instruction, pipeline control, clocks, etc. This cost gets shared by two instructions when they are packed. In addition, the circuit-state overhead current between the two adjacent unpacked instructions (`LAB` and `MSPC`) is eliminated. Since minimizing the total energy consumption is our objective, instructions should be packed under the packing rules, as much as possible.

5 Effect of Circuit State Overhead

The primary components of the models developed in [1, 2] are *base costs* of instructions and *overhead costs* between adjacent instructions. The base current of an instruction is measured by putting several instances of the target instruction in an infinite loop. If a pair of different instructions, say i and j, is put into an infinite loop for measurement, the current is always larger than the average of the base costs of i and j. The difference is called the overhead cost of i and j, and is considered as a measure of the change in circuit state from instruction i to j, and vice-versa. So the total energy consumed by a program is the sum of the total base costs and the total overhead costs, over all the instructions executed.

However, the overhead cost for the above processors only considers the circuit state change caused by adjacent instructions. In the case of the target DSP processor, this can underestimate current, especially for multiply instructions.

Table 1 gives an example program consisting of a sequence of multiply instructions (`MUL:LAB`) followed by `NOP`s. The associated base cost (in mA), and the number of execution cycles is also shown. The overhead cost of instruction pairs are: 1&2: 18.4, 2&3: 18.4, and 1&3: 13.9. The sum of measured current for the four instructions is 204.0 (which equals $I * n$ for the sequence). The sum of the base costs (37.2 + 14.4 + 36.6 + 14.4) and the overhead costs of adjacent instructions (18.4 + 18.4 + 18.4 + 18.4) is only 176.2, which underestimates the actual cost by 13.6%.

The difference, 27.8, in the two estimates actually comes from the circuit state overhead between non-adjacent instructions 1&3. This is due to a special design at the inputs of the multiplier. A latch for each input operand is put between the multiplier and the operand bus to retain the old values until the next multiply instruction is executed. Therefore, the state change at such input latches cannot be accounted for by the overhead of adjacent instructions 1&2 or 2&3. It is given by the overhead of instructions 1&3. So 2 times the overhead of 1&3, (2*13.9mA), can compensate for the above difference, leading to a more accurate estimate[1].

As a result, the new power model needs to include the overhead caused by non-adjacent multiply instructions. Now, this overhead is dependent on the previous and current values of the input latches for each multiply operation. But these values are typically unknown until runtime. So for the purpose of program energy evaluation, the state of the input latches is considered unknown, and an average overhead current penalty is added to the base cost of each multiply instruction. This average value was determined to be 12.5mA for `MUL` (and `MAC`) instructions. So in a way, the above effect is handled by using an enhanced form of base cost for multiply instructions. The enhanced base cost is the base cost as defined earlier, plus an average overhead penalty. While most instructions in the instruction set did not show the same effect, some other instructions involving the ALU data-path did.

6 Power Analysis of DSP Processor

The instruction-level power modeling technique described in Section 5 suggests that accurate current estimation for a program can be obtained if a table that gives the base cost for each instruction, and a table that gives the overhead cost for each instruction pair can be derived. Such tables can be empirically constructed through appropriate experiments using the measurement based power analysis technique. However, there are some practical issues to be considered in this regard. First, the power cost of some instructions can vary depending on the operand value. Extensive experimentation can lead to the development of accurate models for this variation. A practical approximation in this case is to use average costs for these instructions. The average costs are then tabulated. The other issue is one of table size. For processors with rich instruction sets, assigning power costs to all instructions and instruction pairs can lead to large tables. Creation of these tables may require a lot of work. However, it has been observed that instructions can be arranged in classes such that the instructions in a given class have very similar power costs. Instructions with similar functionality tend to fall in the same class. Assigning an average cost to an instruction class can lead to more compact tables. Thus, while having greater detail or resolution in the tables can lead to more accurate cost estimation, for most practical purposes, the use of compact tables suffices.

[1]Because these four instructions are put in an infinite loop for measurement, the overhead will occur twice, between 1 and 3 as well as 3 and 1.

Table 2: Six instruction classes.

class	addressing
LDI	immed → reg (load immediate data to a register)
LAB	mem1 → reg A and mem2 → reg B (transfer memory data to registers A, B)
MOV1	reg1 → reg2 (move data from one register to another)
MOV2	mem → reg, or reg → mem (move data from memory to a register, or from a register to memory)
ASL	reg specified implicitly (add/sub, shift, logic operations in ALU)
MAC	reg specified implicitly (multiply and accumulate in ALU)

Table 3: Average base cost for unpacked instructions.

	LDI	**LAB**	**MOV1**
range	15.8 - 22.9	34.6 - 38.5	18.8 - 20.7
average base	19.4	36.5	19.8

	MOV2	**ASL**	**MAC**
range	17.6 - 19.2	15.8 - 17.2	17.0 - 17.4
average base	18.4	16.5	17.2

For the target DSP processor, the instructions most commonly used in DSP applications were categorized into classes. Six classes were used for unpacked instructions, as shown in Table 2. The principal packed instructions were similarly classified. Extensive current measuring experiments were then conducted to verify for each class the characteristics of current consumption. Furthermore, the effect of different operand values on the variation of current consumption was studied for each class. The average base and overhead costs were also assigned. All these analysis results are discussed in detail in the remainder of this section. Packed and unpacked instructions are discussed separately. A scheduling algorithm that has been developed to use this information for energy reduction will be described in Section 7.

6.1 Unpacked Instruction

Table 3 gives for each unpacked instruction class, the range of base costs for different operand values. The exact operand values are often unknown until runtime. Thus, average values are used during program energy evaluation. These are also shown in Table 3. Since the range of variation in the base costs of each class is reasonably small (less than 10%) for most classes (LDI being the exception), any inaccuracy resulting from the use of averages is limited.

The overhead costs between instructions belonging to different classes are shown in Table 4. The entry in row i and column j gives the overhead cost when an instruction belonging to class j occurs after an instruction belonging to class i, or an instruction belonging to class i occurs after an instruction belonging to class j. This table is symmetric, since the method used for calculating overhead costs assumes that the costs in these two cases are the same. There is a variation in the value of each entry for different operands and for the choice of instructions in each class. This variation

Table 4: Average overhead cost for unpacked instructions.

	LDI	LAB	MOV1	MOV2	ASL	MAC
LDI	3.6	13.7	15.5	6.3	10.8	6.0
LAB		2.5	1.9	12.2	20.9	15.0
MOV1			4.0	18.3	10.5	3.8
MOV2				25.6	26.7	22.2
ASL					3.6	8.0
MAC						12.5

Table 5: Average base cost for packed instructions.

instruction	**ASL**:**LAB**	**ASL**:**MOV1**	**ASL**:**MOV2**
range	34.5 - 38.7	15.7 - 17.4	18.7 - 20.4
average base	36.6	16.6	19.6

instruction	**MAC**:**LAB**	**MAC**:**MOV1**	**MAC**:**MOV2**
range	33.9 - 39.9	15.9 - 18.9	19.0 - 21.2
average base	36.9	17.4	20.1

is again limited, and it is reasonable to use average values. The entries in Table 4 represent the determined average values. The value in the **MAC, MAC** entry represents the overhead that can occur even if the two instructions are non-adjacent, as described in Section 5. An alternative way to look at this case is to use the enhanced base costs of Section 5. The base cost for **MAC** in Table 3 can be increased by 12.5 and the **MAC, MAC** entry in Table 4 can be changed to 0.

An important observation from Table 4 is that there is significant variation across the various entries in the table. This suggests that choosing an appropriate order of instructions can lead to an energy reduction. An algorithm for doing so is described in Section 7.

6.2 Packed Instruction

Table 5 shows for each packed instruction class, the range of base cost variation caused by all possible operand values. Again, the variation is reasonably small (less than 10%) for most classes. An average value is assigned as the base cost, which is also shown in Table 5.

For the overhead cost, experiments showed that except for instructions that have a packed **MAC**, most packed instructions have small ranges of variation. So an average value can be assigned as the overhead cost for these packed instructions. These results are presented in greater detail in [5].

As to the overhead cost of **MAC** instructions, when **MAC** is packed with a data transfer instruction, especially **LAB**, which changes data values in registers A and B used by **MAC** as inputs, significantly wide variation of overhead cost is observed (from 1.4 mA to $33.0mA$). Such wide variation is mainly due to the complex Booth multiplier implemented in the MAC unit.

For a typical DSP application, **MAC**:**LAB** instructions are usually applied to a sequence of data for filter operations, such as $\sum c_i * X_i$. Ideally, the pairwise overhead cost can be used to arrange the data ordering such that the total overhead cost, or the sum of individual pairwise overhead costs, is minimized. But the problem is that X_i is usually not available until execution time. Hence, for our estimation purpose, the average value, $17.2mA$, is used as the overhead cost for **MAC**:**LAB** instructions, due to the unavailability of execution-time operands.

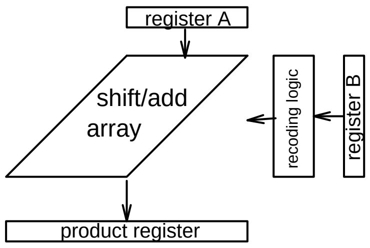

Figure 2: Microarchitecture model for the Booth multiplier.

However, for the purpose of minimization, this single overhead cost value cannot guide the search procedure to a better schedule for a sequence of **MAC**:**LAB** instructions. In any case, for filter applications such as $\sum c_i * X_i$, instruction scheduling of existing code may not be the best alternative. The reason is that the arrival order of operands X_i is determined by the environment of the embedded processor, and is not under the control of a scheduler. Thus, the overall design of the system or algorithm may have to be changed to produce more favorable signal statistics. This may not always be possible. Therefore, under such environmental constraints, in order to still reduce the energy consumption due to **MAC**'s, a more effective technique of local code modification is proposed in Section 6.3, based on exploiting the architecture of the Booth multiplier.

6.3 Operand Swapping for Booth Multiplier

The Booth multiplier implemented in the MAC unit takes the data in registers A and B as operands for fast multiplication. The fundamental idea behind Booth multiplication is to recode B by a so-called "skipping over 1s" technique [6]. For instance, for a 7-digit B value 0011110 that would need four additions of shifted A, it can be recoded to 01000$\bar{1}$0 ($\bar{1}$ denotes -1, for simplicity), which now requires only one addition and one subtraction. However, in the worst case, B may have alternating 1s and 0s, and each bit in B selects a shifted version of A to add or subtract. In order to determine how many additions and subtractions are needed by the Booth multiplier, we can define the *weight* of B value as the number of non-zero digits in its recoded representation. For instance, the weight of 0011110 is 4, while the weight of 01000$\bar{1}$0 is 2.

A simple model of the microarchitecture of the Booth multiplier is depicted in Figure 2. The Booth multiplier does not treat A and B symmetrically. The weight of recoded B determines the number of times A is added or subtracted while generating the product. So if the weight of A is smaller than that of B, we can reduce the number of additions and subtractions by just swapping the operands in registers A and B, which can potentially result in current reduction. Table 6 gives three experiments where swapping the operands of the Booth multiplier reduces current significantly. This observation points out that an effective way to reduce current for **MAC** instructions is to just swap the operands in A and B.

Table 6: Variation of measured current by swapping operands op1 and op2 in registers A and B for **MAC**:**LAB** instructions.

operands		measured current		
op1	op2	op1 * op2	op2 * op1	%saving
`7FFFFF` `000001`	`AAAAAA` `AAAAAA`	58.9	46.9	20.4%
`7FFFFF` `000001`	`666666` `AAAAAA`	68.5	47.9	30.1%
`7FFFFF` `000001`	`AAAAAA` `000001`	65.7	49.1	25.3%

A simple power consumption model based on the microarchitecture model of the Booth multiplier in Figure 2 was empirically derived and validated through extensive current measurement experiments. In this power model, the switching activity of the multiplier is characterized mainly by the contents of registers A and B. Since circuit state is a significant factor for the multiplier, pairs of consecutive values in the registers are considered. For register A, the bit switching between consecutive values is considered, which can determine the complete switching activity in register A and part of the activity in the shift/add array. For register B, two factors are considered. First, the bit switching between consecutive values, and second, the weight of the Booth recodings of the values, which determines the number of additions and subtractions in the shift/add array. Table 7 shows the average current drawn by **MAC**:**LAB** for different characteristics of the pair of consecutive values in A and B. An index (1 to 10, shown in the square parentheses) is assigned to each entry to identify the data characteristics of A and B that the entry represents. For example, entry 8 represents the case where there is high switching between the pair of consecutive values in A, and lot switching between the values in B. In addition, both values in B have high Booth recoding weights. In Table 6 the first pair of data is such an example with such characteristics.

It can be seen from Table 7 that average current for the entries where B has high recoding weights is consistently higher than that in other corresponding entries. Moreover, we can see that entry 9 incurs the highest average current. This is the case where both A and B switch significantly and B has high recoding weights. The second pair of data in Table 6 is an example of such a case. If we swap the two sets of operands in A and B, the characteristics of A and B are now changed. One of the new possibilities is that A still has high switching, but B, which takes the values originally stored in A, can have high switching but low Booth recodings. So it is possible that after swapping, the values of the operands now fall under the case represented by entry 7 in Table 7. Thus, the current drawn may be sharply reduced.

For filter operations such as $\sum c_i * X_i$, the value of the constants c_i is usually known at the time of instruction scheduling. So the scheduler can calculate the weight of the Booth recoding of c_i, and then decide to load c_i into register A, if the recoding weight is high, and into register B, if the recoding weight is low. But the decision about the placement of operands is being made based on the knowledge of the value of just one

Table 8: Current reduction by swapping A and B, which changes the operand characteristics from one entry to another in Table 7.

swapping					
before	after	%saving	before	after	%saving
3	1	16.2%	9	7	26.7%
	3	0.0%		9	0.0%
4	6	23.2%		10	5.4%
	8	-0.7%	10	7	22.5%
5	6	20.5%		9	-5.7%
	8	-4.3%		10	0.0%
8	1	30.1%			
	3	16.6%			

of the operands. Thus, sometimes the wrong decision may be made. However, on the average, determining the placement of operands based on the knowledge of even one operand will lead to current reduction. A systematic investigation was conducted to determine the possible improvements, and the results are shown in Table 8. The known operands are initially assumed to be in register B. If the recoding weight of the value in B is high, the operands are swapped. This means that in case the initial data characteristics fall under the entries in the last 3 columns of Table 7, the operands will be swapped. Table 8 gives the average current reduction when swapping changes the operand characteristics from one entry to another. The columns under the heading "before" show the entries in Table 7 that will result in an operand swap. The column "after" shows the new cases that can arise when the operands in the A and B registers are swapped. The average percentage reduction in the current after operand swapping is shown under the column labeled "% saving". In a few cases there is either no current reduction, or a minor increase. But in a great majority of the cases, we can see that operand swapping can significantly reduce the current. Thus, on the average, the current drawn by **MAC**:**LAB** instructions can be reduced, even though only one operand, for instance, c_i is known at schedule time. Operand swapping is easily achieved by locally modifying the given instruction, e.g., from `MSPC:LAB (X0+1),(X1+1)` to `MSPC:LAB (X1+1),(X0+1)`. In addition, there is no performance or code size penalty associated with it.

7 Low-Power Instruction Scheduling

Based on the power analysis discussed in the previous sections, a low-power scheduling algorithm has been developed. The algorithm is described in greater detail in [5]. The algorithm schedules one basic block of a DSP program at a time. It looks up overhead cost tables and Table 7 to swap operands. Given the register allocation for each basic block, a data flow graph (DFG) is obtained for each block. Then a greedy as-soon-as-possible packing procedure is applied to pack instructions, if data dependency and packing rules allow. So an ALU-type instruction will be packed with a data transfer instruction as soon as the operands of both instructions are ready. The DFG is updated as well, according to the instructions just packed. The goal here is to reduce the number of execution cycles, and therefore, the total energy. Because the exact ex-

Table 7: Average current drawn by **MAC**:**LAB** for different characteristics of consecutive values in A and B.

		B				
		low→low weight		high→high weight		high↔low weight
		low switching	high switching	low switching	high switching	high switching
A	low switching	40.9 [1]	46.4 [2]	48.8 [3]	58.1 [4]	56.1 [5]
	high switching	44.6 [6]	49.9 [7]	58.5 [8]	68.1 [9]	64.4 [10]

Table 9: Comparison of $I * n$ for 5 DSP programs by different scheduling techniques.

benchmark	un_p	p	p+o	p+o+s
$FJex1$	338.0 (1.00)	297.6 (0.88)	256.8 (0.76)	n/a
$FJex2$	474.0 (1.00)	380.7 (0.80)	342.9 (0.72)	n/a
LP_FIR60	6987.5 (1.00)	3731.0 (0.53)	n/a	3100.5 (0.44)
$IIR4$	1228.5 (1.00)	906.0 (0.74)	822.0 (0.67)	772.0 (0.63)
$FFT2$	1174.4 (1.00)	1133.9 (0.97)	1087.5 (0.93)	(0.89) 1046.9

ecution order of the packed and unpacked instructions is not determined until the following scheduling step, the effect of overhead cost is not considered during instruction packing.

Then the scheduler attempts to construct an execution order for the instructions that obey data dependency, while minimizing the total overhead cost. The implementation for this step is based on the popular *list scheduling* algorithm [7], with the overhead cost as the objective function to be minimized. Pipeline stall conditions due to resource constraints or data hazards can also be handled during list scheduling, to avoid penalties due to extra cycles.

Finally, if the code contains multiply operations where at least one of the operands is known, e.g., filter operations like $\sum c_i * X_i$, where c_i's are available, the scheduler can swap the operands based on Table 7 to further reduce the current.

8 Experimental Results

Table 9 shows the experimental results for five DSP programs to demonstrate the energy reductions possible by the above scheduling method. Column 1 lists the name of each benchmark program. The remaining columns show the energy comparisons by applying different scheduling techniques: `up_p` for the original unpacked code, `p` for packing alone, `p+o` for both packing and overhead cost reduction, and `p+o+s` for the combined application of packing, overhead cost reduction, and operand swapping.

The first two programs $FJex1$ and $FJex2$ were real Fujitsu applications for vector preprocessing, where no **MAC** instructions are used (so operand swapping is not applicable). The third program LP_FIR60 is a length-60 linear phase FIR filter; the fourth program $IIR4$ is a fourth-order direct form IIR filter; and the fifth program $FFT2$ is a radix-2 decimal-in-time FFT butterfly. The last three programs were taken from the TMS320 embedded DSP examples in [8] and translated into native code for our target processor. For each benchmark program, the product $I * n$ (which is proportional to energy) is given. The values in the parentheses are the relative values when the products in column `un_p` are normalized to 1.

The results show that about 3% to 47% energy reduction can be achieved by instruction packing alone. The reason that $FFT2$ has only 3% reduction is due to a certain data dependency imposed by the unpacked code that prevents a **MOV2** instruction from being packed with the preceding **ASL** instruction. The **ASL** instruction generates a new register value for **MOV2** to transfer in the next cycle. After packing, the list scheduling algorithm based on overhead cost reduction can further reduce energy by 4% to 14% for the packed codes. In the cases of $IIR4$ and $FFT2$, operand swapping is applicable, and additional 7% and 4% energy can still be saved, respectively. So the overall energy reduction is seen to be 11% to 56% if the source code is originally unpacked, and 8% to 17% if it is originally packed.

Acknowledgment: The authors would like to thank Fumiyasu Hirose, Hirohisa Gambe, and Hirokazu Fukui of Fujitsu Ltd. for their assistance on experiment setup and technical discussion.

References

[1] V. Tiwari, S. Malik, and A. Wolfe. "Power analysis of embedded software: A first step towards software power minimization". In *IEEE Trans. VLSI Systems*, Dec. 1994.

[2] V. Tiwari and Mike T.-C. Lee. "Power analysis of a 32-bit embedded microcontroller". To appear in *Asia and South Pacific DAC*, Aug. 1995.

[3] V. Tiwari, S. Malik, and A. Wolfe. "Compilation techniques for low energy: An overview". In *Proc. Symp. on Low Power Electronics*, Oct. 1994.

[4] C.-L. Su, C.-Y. Tsui, and A. M. Despain. "Saving power in the control path of embedded processors". In *IEEE Design & Test of Computer*, winter 1994.

[5] Mike T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita. "Power analysis and low-power scheduling techniques for embedded DSP software". *Technical Report FLA-CTM-02*, Fujitsu Labs. of America, 1995.

[6] Kai Hwang. *Computer Arithmetic - Principles, Architecture, and Design.* John Wiley & Sons, 1979.

[7] Mike Johnson. *Superscalar Microprocessor Design*, chapter 10. Basic Software Scheduling. Prentice Hall, 1990.

[8] Texas Instruments. *Digital Signal Processing Applications - Theory, Algorithm, and Implementations.* 1986.