Power Analysis and Minimization Techniques for Embedded DSP Software

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

***Abstract*—Power is becoming a critical constraint for designing embedded applications. Current power analysis techniques based on circuit-level or architectural-level simulation are either impractical or inaccurate to estimate the power cost for a given piece of application software. In this paper, an instruction-level power analysis model is developed for an embedded digital signal processor (DSP) based on physical current measurements. Significant points of difference have been observed between the software power model for this custom DSP processor and the power models that have been 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 processor has special architectural features that allow dual-memory accesses and packing of instructions into pairs. The energy reduction possible through the use of these features is studied. The on-chip Booth multiplier on the processor is a major source of energy consumption for DSP programs. A microarchitectural power model for the multiplier is developed and analyzed for further power minimization. In order to exploit all of the above effects, a scheduling technique based on the new instruction-level power model is proposed. Several example programs are provided to illustrate the effectiveness of this approach. Energy reductions varying from 26% to 73% have been observed. These energy savings are real and have been verified through physical measurement. It should 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 remain unchanged.**

***Index Terms*—Embedded DSP software, embedded system design, power analysis, power optimization.**

I. 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 or application specific instruction-set processors (ASIP's) that have been specially designed for a certain class of algorithms. In most of the embedded applications, such as cellular phone or portable electronic devices, power becomes an important constraint in the design specification. 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. At present, accurate power estimation tools are available only for the lower levels of the design-at the circuit level and to a limited extent at the gate level. For an embedded processor, circuit-level or gate-level simulation is slow and impractical to evaluate the power consumption of software. These techniques often cannot even be applied due to lack of circuit-level and gate-level information of the embedded processor. In [3]–[5] recourse is taken to architectural-level power simulation. This takes the software instructions as stimulus and sums up current of the active modules in the processor for each simulation cycle. Here, the current assigned to each module is the cost of being active, and is usually constant—regardless of the operand value, circuit state, and the correlation with the activities of other modules. In [6], architectural power analysis based on stochastic data modeling is proposed for greater accuracy for ASIC digital signal processor (DSP) applications. However, due to the higher levels of abstraction in these works, the power estimates are not very accurate. Further, lower level internal details of the processors are still needed, in order to assign power costs to modules.

All of the above problems can be overcome if the current drawn by the CPU during the execution of programs is physically measured. An instruction level power analysis technique based on physical measurements has recently been developed by Tiwari *et al.* [1]. This paper discusses the application of this technique for the power analysis of a Fujitsu embedded DSP processor. This processor, referred to as the target DSP processor from here on, is used in several Fujitsu embedded applications, and is representative of a large class of DSP processors. The analysis results are used to derive an instruction-level power model that makes it possible to evaluate the power cost of programs that run on the target DSP processor. The results of the analysis also motivate several software power minimization techniques that exploit the power consumption characteristics of the processor. These are also described in the paper.

The instruction-level analysis technique had earlier been applied to two large general-purpose commercial microprocessors [1], [2]. Some points of significant difference between the power models of these two general-purpose processors and the target DSP processor have been identified. In particular, the effect of circuit state change is found to be more marked in terms of power consumption for the target DSP processor than for the previous two processors. This effect had limited impact for the other processors, and it was felt that this may be characteristic of large general-purpose processors, as opposed to smaller, specialized processors like DSP's. The significant impact of circuit state in the case of the target DSP processor suggests that an appropriate scheduling of instructions can lead

Manuscript received July 31, 1996.

M. T.-C. Lee and M. Fujita are with Fujitsu Laboratories of America, Santa Clara, CA 95054 USA.

V. Tiwari and S. Malik are with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA.

Publisher Item Identifier S 1063-8210(97)00737-3.

to a reduction in the power cost of programs [7]. Furthermore, it is seen that for certain instructions that involve the ALU datapath, even nonadjacent instructions can cause changes in the circuit state that contributes to significant power consumption. This effect was not observed in the previous work. If this effect is not considered for the target DSP processor, power for certain programs can be underestimated—e.g., by about 13.6%—which is described later.

The issue of scheduling for power minimization has been explored to a limited extent in earlier works [8], [9]. In [8], the study shows that faster programs consume less energy. So optimizing software performance through instruction scheduling can minimize the energy consumption. In [9], only the power consumed by the controller of a processor is targeted for minimization by instruction scheduling. The power cost of different instruction schedules is estimated by counting transitions on an RTL level model of the control-path. This is a rough measure of the actual power cost. Furthermore, the increase in energy cost due to longer schedules is not considered. The instruction scheduling technique that we propose overcomes the previous limitations since it is based on actual energy costs obtained through physical measurements, and is therefore more effective,

The DSP processor has some special architectural features that were originally provided to reduce the number of cycles for programs that can utilize the features. However, our analysis shows that these feature are also very effective for reducing the energy cost of programs. The first feature allows double data transfers from different memory banks to registers in one cycle [10], and the other packs two instructions into a single code-word. Automated techniques for effectively exploiting these features for energy reduction are presented. Another avenue for power reduction is based on the observation that the on-chip Booth multiplier is a major source of energy consumption for DSP programs, This motivates the development and analysis of a microarchitectural power model for the multiplier. Based on this model, an effective technique for local code modification by operand swapping is proposed to further reduce power consumption.

Finally, a energy minimization methodology is presented to collectively apply the above techniques to any given piece of code. Experimental results on several example DSP programs show energy reductions ranging from 26 to 73%. The experimental set-up used allows for immediate verification of results. Thus, the energy savings reported in the paper have been physically validated. 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.

This paper is organized as follows. Section II highlights the architectural features of the target processor relevant to our study on power analysis and optimization. Section III explains the experiment setup for current measurement. Section IV develops an instruction-level power model for the target DSP processor based on physical current measurement, and highlights its major differences from previous models. Section V discusses the proposed energy optimization techniques based on memory bank assignment, instruction packing, instruction scheduling, and operand swapping for the Booth multiplier. Section VI presents the experimental results, and Section VII summarizes the contributions and future directions of this work.

II. Architecture of the Target DSP Processor

The target DSP processor used for our study is a Fujitsu 3.3 V, 0.5 μm, 40 MHz CMOS processor. It implements several special architectural features to support embedded DSP applications. Some of these features are usually not seen in general-purpose microprocessors. The basic architectural features of the processor that are relevant to the rest of the paper are listed below. The impact on power consumption of some of these will be studied in detail when the features are discussed further in the following sections.

- Reduced number of pipeline stages is two. In the first stage, the instruction is fetched and decoded; in the second stage, the necessary operands are accessed, the result is computed and finally written back.
- Limited number of data registers is four. These are 24 b registers A, B, C, and D.
- Eight index registers to support sequential access to data arrays in the memory banks, a data structure commonly found in DSP applications.
- A fast MAC (Multiply-and-ACcumulate) unit. A MAC operation is completed in one cycle by using a Booth multiplier. Power analysis and optimization for the MAC unit is discussed in Section V-D.
- Simple instruction set restricted by the special DSP architecture. For example, the Booth multiplier always takes the operands from registers A and B. Explanation and classification of the instruction set will be given in Section IV-B.
- Power management for the Booth multiplier. The two operands of the Booth multiplier are latched to retain their old values to reduce unnecessary switching in the multiplier. Its effect on the proposed instruction-level power model is discussed in Section IV-A.
- Two on-chip memory banks—RAMA and RAMB. If two data operands are in different banks, they can be fetched simultaneously into registers by a double-transfer instruction. A low-energy memory bank assignment technique is proposed in Section V-A to exploit this feature.
- Packed instruction. In general, an ALU-type instruction and a data transfer instruction can be packed into a single instruction codeword for simultaneous execution. Energy optimization through the use of this feature is discussed in Section V-B.

III. Current Measurement

The average power P consumed by a processor while running a certain program is given by $P = I * V_{dd}$, where I is the average current and V_{dd} is the supply voltage. The energy consumed by a program, E, is given by $E = P * T$, where T is the the execution time of the program. This in turn is given by $T = N * \tau$, where N is the number of clock cycles and τ is the clock period.

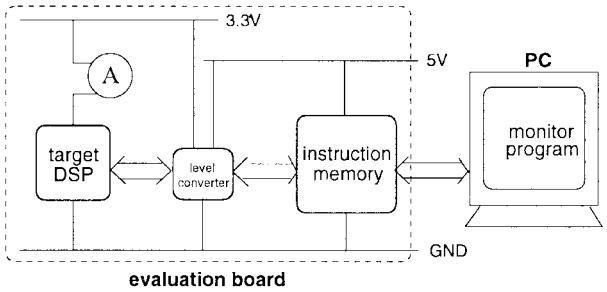

Fig. 1. Experiment setup to measure the current drawn by the target DSP using an ammeter denoted by the circle A.

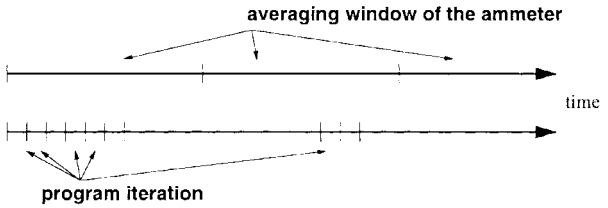

Fig. 2. Obtaining steady current readings from the digital ammeter.

Since the applications of the target DSP processor run on the limited energy available in a battery, energy consumption is the focus of attention. While we will attempt to retain the distinction between the terms "power" and "energy," the "power" is often used to refer to "energy," in adherence to common usage. Now, V_{dd} and τ are known and fixed. Therefore, E is proportional to the product of I and N. Given the number of execution cycles, N, for a program, we just need to measure the average current, I, to calculate E. The product $I*n$ (with I in mA) is the measure used to compare the energy cost of programs in this paper.

The current measurement setup is illustrated in Fig. 1. This processor is part of a personal computer evaluation board with several 3.3 V/5 V level converters and instruction memory operating at 5 V. The DSP chip 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 two on-chip memory banks of the DSP processor. The current drawn by the DSP processor is measured through a standard off-the-shell, dual-slope integrating digital ammeter (indicated by the circle A in Fig. 1), which is connected between a 3.3 V power supply and the V_{dd} pins of the target DSP chip. The layout of the particular evaluation board makes it difficult to isolate the power connections to the external memory chips, and thus the measurement results do not include this current.

If a program completes execution in a short time, a current reading cannot be visually obtained from the ammeter. To overcome this, the programs being considered are put in infinite loops and current readings are taken. The current consumption in the CPU varies in time depending on what instructions are being executed. But since the chosen ammeter averages current over a window of time (100 ms), if the execution time of the program is much less than the width of this window, a stable reading will be obtained. This is illustrated in Fig. 2.

The main limitation of this approach is that it will not work for longer programs, since the ammeter may not show a stable reading. However, this is not a limitation for the development of an instruction-level power model, since short instruction sequences are all that is needed. To determine the base cost of a given instruction, a loop consisting of a sufficient number of instances of the instruction, for example, 200, is needed. Similarly, to assign energy costs to specific inter-instruction effects like circuit-state, pipeline stalls, etc., appropriate short instruction sequences are needed. This is discussed in greater detail in [1]. For these purposes, the inexpensive approach of using a digital ammeter works very well. It should be stressed that the main concepts described in this paper are independent of the actual method used to measure average current. The results of the above approach have been validated by comparisons with other current measurement setups. But if sophisticated data acquisition based measurement instruments are available, the measurement method can be based on them, if so desired.

IV. Power Analysis for the Target DSP Processor

A comprehensive instruction-level power analysis of the target DSP processor has been performed using the current measurement technique discussed in the previous section. The salient results of the analysis are described in this section. We begin by describing an inter-instruction effect that was not evident in the case of previous power models [1], [2]. This has to do with the effect of circuit state in the special design of the on-chip multiplier. An enhanced power model is proposed to account for this effect. Subsequently, the other parameters of the complete instruction-level power model of the processor are described.

A. Effect of Circuit State Overhead

Instruction-level power analysis models for the Intel 486DX2 and the Fujitsu SPARClite have been developed earlier as described in [1], [2]. The primary components of these models 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. The results for our target DSP processor show that this model can underestimate current, especially for multiply instructions.

Table I gives an example program consisting of a sequence of packed instructions (`MUL:LAB`) followed by `NOP`'s. The packed instruction `MUL:LAB` multiplies registers A and B while in the same cycle transferring two new data from

TABLE I
AN EXAMPLE OF A SEQUENCE FOUR INSTRUCTIONS WHERE THE OVERHEAD COST BETWEEN 1 AND 3 CANNOT BE IGNORED

number	instruction	base	overhead		cycles
1	`MUL:LAB (X0+1),(X1+1)`	37.2	(1&2)	18.4	1
2	`NOP`	14.4	(2&3)	18.4	1
3	`MUL:LAB (X0+1),(X1+1)`	36.6	(3&4)	18.4	1
4	`NOP`	14.4	(4&1)	18.4	1
total:		102.6	+	73.6	= 176.2

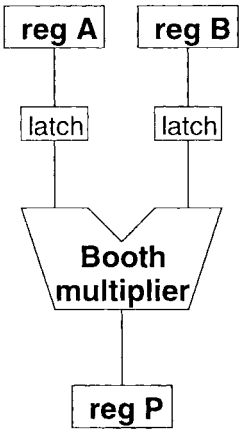

Fig. 3. Latches at the inputs of the Booth multiplier retain the old operands to reduce unnecessary switching.

memory to registers A and B respectively. The associated base cost, the overhead cost of instruction pairs, and the number of execution cycles are listed in Table I. 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 numbers actually comes from the circuit state overhead between nonadjacent instructions 1 and 3. This is due to a special design feature at the inputs of the multiplier as illustrated in Fig. 3. 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 and 2 or 2 and 3. It is given by the overhead of instructions 1and 3. So 2 times the overhead of 1 and 3, $(2*13.9$ mA), 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 nonadjacent 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 12.5 mA for `MUL` 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. It can also be expected that other processors that utilize similar designs may also exhibit a similar effect.

[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.

B. Instruction-Level Power Model

The instruction-level power modeling technique described in Section IV-A suggests that accurate current estimates 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.

For the target DSP processor, the instructions most commonly used in DSP applications were categorized into classes. Six classes were used for unpacked instructions. The principal packed instructions are similarly classified. The instructions in the same class have similar functionality and activate similar parts of the CPU. Hence, they have similar characteristics with regards to the current drawn. These six instruction classes are listed in Table II, where the addressing mode and the corresponding functionality is also provided.

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

TABLE II
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 III
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 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 V-C.

1) Base/Overhead Cost of Unpacked Instruction: Table III 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 III. 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 IV. 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 is again limited, and it is reasonable to use average values. The entries in Table IV represent the determined average values. The value in the **MAC, MAC** entry represents the overhead that can occur even if the two instructions are nonadjacent, as described in Section IV-A. An alternative way to look at this case is to use the

TABLE IV
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 V
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

enhanced base costs of Section IV-A. The base cost for **MAC** in Table III can be increased by 12.5 and the **MAC, MAC** entry in Table IV can be changed to 0.

An important observation from Table IV is that there is significant variation across the various entries in the table. This is in contrast to previous power models. In the power models for the the large general-purpose processors, the variation in the circuit state overhead for different instruction pairs was limited. The use of a constant value to account for this overhead was justified in those cases, but will be inaccurate for the target DSP. The data in the table also suggests that choosing an appropriate order of instructions can lead to an energy reduction. A scheduling algorithm for doing so is described in Section V-C.

2) Base/Overhead Cost of Packed Instruction: Table V 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 V. An interesting observation is that the base cost of a packed instruction that has a data transfer instruction as a component, is very close to the base cost of the unpacked data transfer instruction alone (cf. Tables III and V). For example, the base cost of **ASL**:**LAB** is 36.6 mA, very close to the base cost of **LAB** alone.

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. The overhead costs of a few packed instructions commonly found in our DSP software are listed in Table VI.

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 current variation is observed. Such wide variation is mainly due to the complex Booth multiplier implemented in the MAC unit. Table VII shows what happens when adjacent **MAC:LAB** instructions use different data. Thus, different operands are loaded into registers A and B

TABLE VI
AVERAGE OVERHEAD COST FOR SEVERAL COMMONLY USED PACKED INSTRUCTIONS

adjacent instructions		overhead
LDI,	**ASL:LAB**	23.8
LDI,	**ASL:MOV1**	19.6
LDI,	**ASL:MOV2**	20.0
MOV1,	**ASL:LAB**	14.7
MOV2,	**ASL:LAB**	27.1
MOV2,	**ASL:MOV2**	19.3
MAC,	**ASL:LAB**	27.9
ASL:LAB,	**ASL:MOV1**	27.1
ASL:LAB,	**MAC:MOV1**	29.3
ASL:MOV2,	**ASL:MOV2**	18.7

TABLE VII
VARIATION OF OVERHEAD COST FOR **MAC**: **LAB**

reg A * reg B	overhead
EFFFFF * FFFFFF	1.4
EFFFFF * EFFFFF	
7FFFFF * 555555	17.3
7FFF00 * 555666	
AAAAAA * 555555	33.0
555555 * AAAAAA	

for multiplication in adjacent instructions. The variation in pairwise overhead cost is seen to be in the range of 1.4 to 33.0 mA, as depicted by three cases shown.

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 given in Table III 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, 17.2 mA, is used as the overhead cost for **MAC**: **LAB** instructions, due to the unavailability of execution-time operands.

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 the next section, based on exploiting the architecture of the Booth multiplier.

3) Estimation Example: A length-60 linear phase FIR filter example adapted from a TMS320 program [11] is used to validate our instruction-level power analysis model. This filter example consists of one basic block and finishes in 66 cycles

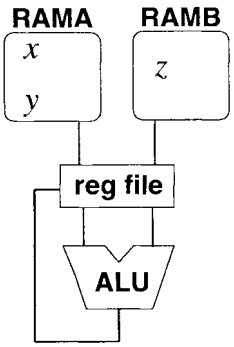

Fig. 4. Datapath of the DSP processor. Transferring to the register file requires two cycles (two MOV's) for x and y, and only one cycle (one LAB) for x and z.

TABLE VIII
COMPARISON OF ENERGY CONSUMED BY SINGLE-TRANSFER AND DOUBLE-TRANSFER INSTRUCTIONS

		2 MOV's	1 LAB	
regA	regB	$I(mA)$	$I(mA)$	%E saving
$000000	$FFFFFF	36.2	35.8	50.55%
$AAAAAA	$CCCCCC	32.0	35.8	44.06%
$FFFFFF	$FFFFFF	25.8	33.8	34.50%

by using packed instructions. The average measured current is 57.4 mA, while the estimated value by the proposed power model is 51.4 mA, with about 10.5% error.

V. ENERGY MINIMIZATION FOR THE TARGET DSP PROCESSOR

Based on the power analysis of the target processor, this section proposes several effective energy minimization techniques for embedded DSP software. The first two techniques, memory bank assignment and instruction packing, exploit the architectural features of dual-memory transfers and packed instructions. These minimize energy by reducing the program execution cycles. Then an instruction scheduling algorithm is introduced to reduce the circuit state overhead cost. In addition, since the on-chip Booth multiplier is a major source of energy consumption, a microarchitectural power model is developed and an energy minimization technique based on swapping the operands is proposed. The above two techniques achieve energy reduction by reducing the average current, without affecting the number of execution cycles. All reported current values are obtained by the current measurement technique described in Section III.

A. Memory Bank Assignment for Low Energy

Section II mentioned that the target DSP processor has two on-chip data memory banks RAMA and RAMB, as depicted in Fig. 4. Each of these can supply data to the register file for an ALU operation, in the same cycle, by a double-transfer instruction—LAB. If these two operands are stored in the same memory bank, two single-transfer instructions MOV's are needed instead. This takes two cycles, one for each transfer. Table VIII shows the average current for these alternatives in Columns 3 and 2, respectively.

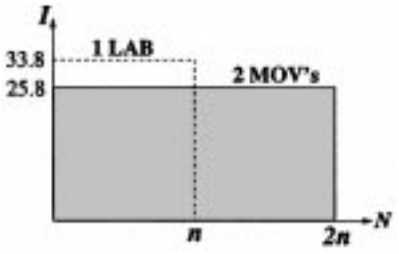

Fig. 5. Comparison of total energy consumption by one LAB and two MOV's.

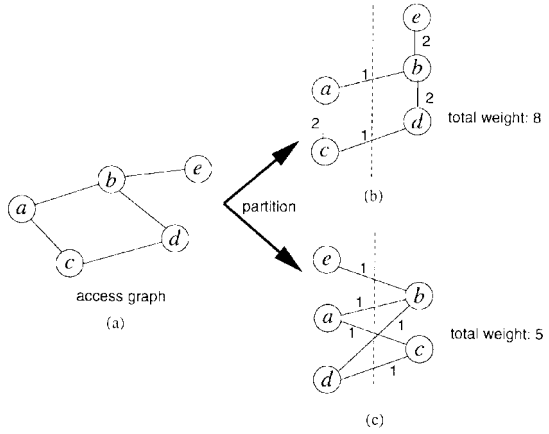

Fig. 6. (a) An access graph for five variables and (b) and (c) represent two possible partitions with different costs.

It is possible to save software energy through the use of LAB, which takes half of the execution time of two MOV's. As seen from Table VIII, the average current for a LAB is the same or marginally higher than that for a sequence of two MOV's. However, since it takes only half the cycles, the double-transfer instruction takes about half the energy, as shown in Column 4. Fig. 5 depicts this graphically for the third entry in Table VIII. The area under the solid and dotted curves is proportional to the energy cost of the two MOV's and one LAB, respectively, since energy is proportional to $I * N$.

Although both types of instructions perform the same function, the energy consumed by two MOV's is always larger than that by a single LAB. This may have to do with the fact that the large power cost associated with the clock, instruction fetch, control, etc., gets shared by the two data transfers when they execute together in one cycle. In addition, the power cost associated with the change in circuit state between the two MOV's is also eliminated. Therefore, in order to reduce energy consumption, variables in an embedded program should be assigned to memory to allow maximal use of double-transfer instructions. We formulate this memory allocation problem as a variable partitioning problem. A simulated-annealing algorithm is proposed next as an efficient solution for the problem.

1) Simulated Annealing for Memory Bank Assignment: Given a basic block of an embedded DSP program, the two operands needed by each ALU operation are first determined. An **access graph** is then constructed where each node v_i corresponds to a variable i in the program. For each ALU operation that requires two variables, i and j, there is a corresponding edge $\langle v_i, v_j \rangle$ in the graph. Any two way partition of this graph represents a valid memory allocation, where the variables corresponding to the nodes in each partition are allocated to the same memory bank.

Given a two-way partition of the access graph, we assign a *cost* value to the partition, which is the number of cycles to transfer data from memory banks to registers. That is, for each edge $\langle v_i, v_j \rangle$, if variables i and j are assigned in

```
SA(G):
{       input:  G: access graph
        T = initial temperature;
        π = initial_2_way_random_partition(G);
        cost = Σ edge cost in π;
        while (!frozen()) {
                while (!equilibrium()) {
                        π' = next_state(π);
                        cost' = Σ edge cost in π';
                        if (accept(cost,cost')) {
                                π = π';
                                cost = cost';
                        }/*else, keep old π and cost*/
                }
                T = update(T);
        }
}
```

Fig. 7. Simulated annealing algorithm for least-cost access graph partitioning.

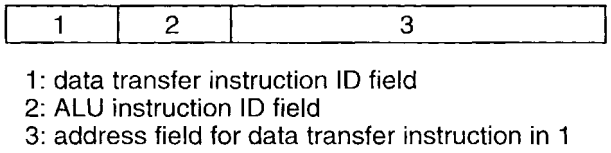

Fig. 8. Packed instruction format.

TABLE IX
CURRENT DRAWN BY PACKED OR UNPACKED INSTRUCTIONS

reg A * reg B	packed (mA)	unpacked (mA)
AAAACC * FFFF00 CCA0A0 * 898989	65.1	60.4
000000 * 000000 0FFFFF * 0FFFFF	53.3	53.3

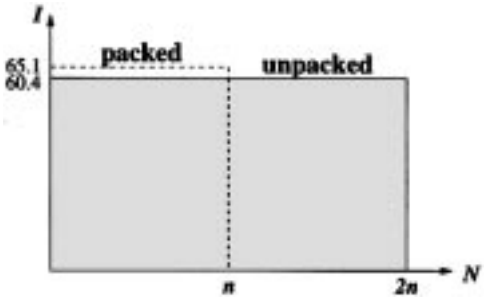

Fig. 9. Comparison of total energy consumed by packed and unpacked instructions.

```
list_scheduling(β,DFG,Γ_overhead):
{
    ctr := 0; /* cycle counter */
    ready_list := new_ready(β,DFG); /* instructions in
        β without predecessors in DFG */;
    β' := β;
    while (ready_list ≠ ∅) {
        inst := lowest_overhead(β',ready_list,Γ_overhead);
        ready_list := ready_list - {inst};
        β' := assign_cycle(β',inst,ctr);
        update_ready_info(DFG);
        ready_list := ready_list ∪ new_ready(β',DFG);
        ctr++;
    }
    return(β');
}
```

Fig. 10. List scheduling algorithm to reduce overhead cost.

TABLE X
BOOTH MULTIPLIER RECODING TABLE FOR $A * B$

B		version of A selected
bit i	bit i-1	by bit i of B
0	0	0 * A
0	1	1 * A
1	0	-1 * A
1	1	0 * A

different memory banks, 1 cycle is needed by a double-transfer LAB; otherwise, two cycles are needed by two single-transfer MOV's. Fig. 6(a) shows an example of an access graph for five variables. Fig. 6(b) and (c) show two possible partitions, which take eight and five cycles to transfer data, respectively. The problem of energy optimization now reduces to finding the partition with the least cost.

An algorithm SA is proposed in Fig. 7 to find the least cost partition. The algorithm is based on the standard formulation for *simulated annealing* [12]. Given an initial random partition of G, SA iteratively generates a new partition until the stopping criteria `frozen` and `equilibrium` are met. The cost of any given partition is determined by summing up the costs over all edges. The function `next_state` generates a new partition by allowing movement of a single node as well as swapping of two nodes. The function `accept` determines at each stage of the algorithm, if the new partition should be accepted or not.

It should be noted that the dual-memory transfer feature is not unique to the Fujitsu DSP, but is also provided by several other popular DSP processors, e.g., the Motorola 56 000 series. The above observations are likely to be valid for these other processors too. In addition, recent memory allocation techniques developed from the point of view of improving performance [13] can also find application for energy reduction.

B. Instruction Packing for Low Energy

As discussed in Section II, the target DSP processor provides the capability of packing an ALU-type instruction and a data transfer instruction into a single instruction codeword

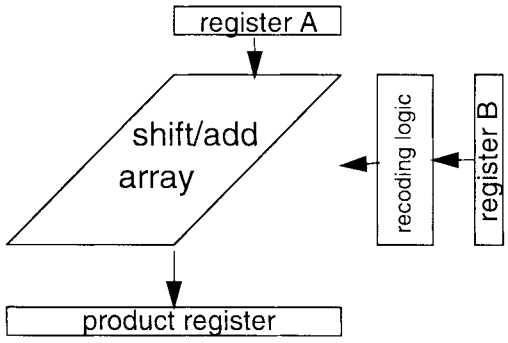

Fig. 11. Microarchitecture model for the Booth multiplier.

TABLE XI
VARIATION OF MEASURED CURRENT BY SWAPPING OPERANDS OP1 AND OP2 IN REGISTERS A AND B FOR **MAC**: **LAB** INSTRUCTIONS

	operands		measured current		
no.	op1	op2	op1 * op2	op2 * op1	%saving
1	7FFFFF	AAAAAA	58.9	46.9	20.4%
	000001	AAAAAA			
2	7FFFFF	666666	68.5	47.9	30.1%
	000001	AAAAAA			
3	7FFFFF	AAAAAA	65.7	49.1	25.3%
	000001	000001			

for simultaneous execution. This feature is called "instruction packing." Fig. 8 shows the packed instruction format. Therefore, if data dependency and the packing rules allow, two instructions can be assigned to the same execution cycle and packed into a single instruction. The single packed instruction represents the same functionality as the sequence of two unpacked instructions. But, interestingly, 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 observation is illustrated by the following example where MSPC, a multiply instruction, and LAB, a load instruction, can either execute as unpacked instructions for a total of two cycles, or as a single packed instruction that executes in just one cycle.

The current drawn by the instructions in packed and unpacked format is compared for two sets of data operands in Table IX. As can be seen from the results, the average current drawn for the packed instructions is only marginally higher than for the unpacked instructions. However, the unpacked instructions complete in twice the number of cycles as the packed instructions, so the total energy consumed by the unpacked instructions is much larger, about twice as much as the packed instructions. Fig. 9 illustrates this comparison graphically for the first set of operands. The area under the graph, which is given by $I * N$, is proportional to the total energy consumption.

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 instruction, which is independent of

TABLE XII
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 XIII
CURRENT REDUCTION BY SWAPPING A AND B, WHICH CHANGES THE OPERAND CHARACTERISTICS FROM ONE ENTRY TO ANOTHER IN TABLE XII

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%			

the functionality of the instruction and independent of whether the instruction is packed or not. 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. A greedy *as-soon-as-possible* (ASAP) packing algorithm has been implemented, which selects the next available instruction in a program as a candidate for packing, if data dependency and packing rules allow.

C. Instruction Scheduling for Low Power

As can be seen from the results in Section IV, the circuit state overhead cost has significant variation across different instruction pairs. Thus, different instruction schedules for the same program can consume different power. This suggests that it is possible to reduce the power cost of a program by an appropriate scheduling of instructions. An automated instruction scheduler has been designed that can minimize the total circuit state overhead cost for a program. It looks up the overhead cost tables and chooses a good instruction schedule without violating data dependencies. The implementation for the scheduler is based on the popular *list scheduling* algorithm [14], with the overhead cost as the objective function to be minimized. The list scheduling algorithm for overhead cost minimization is showed in Fig. 10. At each cycle, a ready list is maintained for the instructions whose operands become available. Then an instruction in the ready list with the lowest overhead cost is selected to be scheduled at the current cycle. Pipeline stall conditions due to resource or data hazards can be verified as well during such instruction selection, to avoid penalties due to extra cycles. The scheduled instruction is then deleted from the ready list, and the new ready instructions arising from the new schedule are added to the ready list.

D. Operand Swapping for the Booth Multiplier

We have found that in typical DSP applications the multiplier in the MAC unit is usually a major source of power consumption. This is because of its complex design, which leads to a large current cost. This is compounded by the frequent usage of filter operations such as $\sum c_i * X_i$ in DSP applications. This section focuses on power analysis of the on-chip Booth multiplier, and proposes a power reduction technique based on swapping the multiplication operands.

1) Power Model of the Booth Multiplier: The Booth multiplier implemented in the MAC unit takes the data in registers A and B as operands for fast multiplication. Without going into the details of the Booth multiplication algorithm, the fundamental idea behind it is to recode B by a so-called "skipping over 1's" technique [15]. Table X shows the basic recoding scheme employed by the Booth algorithm. The motivation for recoding B is that in cases where B has its 1s grouped into a few contiguous blocks, only a few versions of A need to be added/subtracted to generate the product. For instance, for a 7-digit B value 0 011 110 that would need four additions of shifted A, it can be recoded to $0100\bar{1}0$ by Table X ($\bar{1}$ denotes -1, for simplicity), which now requires only one addition and one subtraction. However, in the worst case, B may have alternating one's and zeroes, 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 nonzero digits in its representation. For instance, the weight of 0 011 110 is 4, while the weight of $0100\bar{1}0$ is 2.

A simple model of the microarchitecture of the Booth multiplier is depicted in Fig. 11. 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 XI 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.

A simple power consumption model based on the microarchitecture model of the Booth multiplier in Fig. 11 was

TABLE XIV
COMPARISON OF $I * n$ (WHICH IS PROPORTIONAL TO ENERGY) FOR 5 DSP PROGRAMS BY DIFFERENT MINIMIZATION TECHNIQUES (ORG_ un_p: ORIGINALLY UNPACKED; M: MEMORY BANK ASSIGNMENT; P: PACKING; O: REDUCING OVERHEAD COST; S: SWAPPING OPERANDS; n/a: NOT APPLICABLE; RED%: REDUCTION PERCENTAGE COMPARED WITH org_ un_p).

benchmark	org_un_p	m	(red%)	m+p	(red%)	m+p+o	(red%)	m+p+o+s	(red%)
ex	371.8	338.0	(9%)	297.6	(20%)	256.8	(31%)	n/a	
LP_FIR60	11494.8	7043.4	(39%)	3788.4	(67%)	n/a		3148.2	(73%)
IIR4	1613.5	1228.5	(24%)	906.0	(44%)	822.0	(49%)	772.0	(52%)
FFT2	1414.8	1174.4	(17%)	1133.9	(20%)	1087.5	(23%)	1046.9	(26%)

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 XII 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 low switching between the values in B. In addition, both values in B have high Booth recoding weights. In Table XI the first pair of data is an example with such characteristics.

2) Power Reduction by Operand Swapping: It can be seen from Table XII 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 XI 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 XII. 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 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 XII. 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 three columns of Table XII, the operands will be swapped. Table XIII 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 XII 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.

```
energy_optimization(R,Γ_overhead,Γ_swap,β):
    input: R: packing rules;
           Γ_overhead: overhead cost table (as derived in
                Tables IV and VI);
           Γ_swap: current reduction table for operand
                swapping (as derived in Table XII);
           β: basic block where instruction selection and
                register allocation are done;
{
    DFG := data flow graph of β;
    1: β' := SA_memory_bank_assignment(β,DFG);
    2: (β', DFG') := ASAP_packing(β',DFG,R);
    3: (β') := list_scheduling(β',DFG',Γ_overhead);
    if (β' has filter operation Σ c_i * X_i and c_i's are
                available)
    4:      β' := operand_swapping(β',c_i,Γ_swap);
    return(β');
}
```

Fig. 12. Overall methodology for software energy minimization.

E. Overall Software Energy Minimization Methodology

Based on the previous discussion, an overall energy optimization methodology is summarized in Fig. 12. It takes one

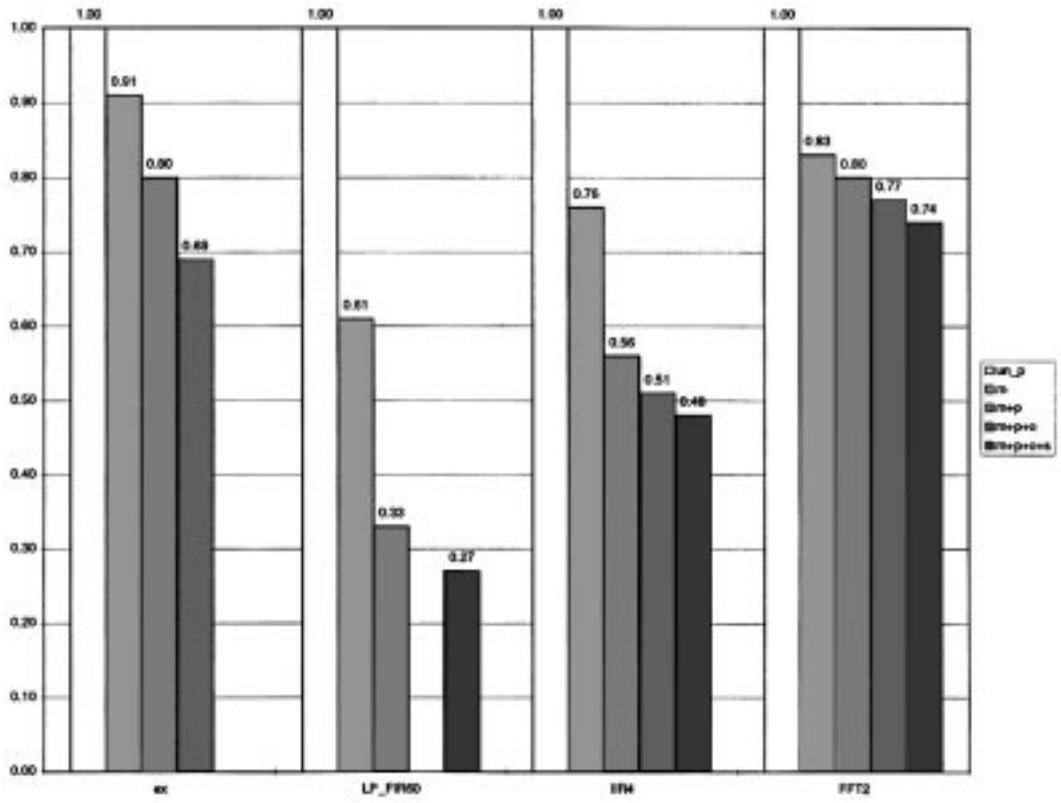

Fig. 13. Energy percentages for the values in Table XIV relative to that of original unpacked code (`org_un_p`) normalized to 1.00.

basic block of a DSP program at a time and assumes that instruction selection and register allocation have already been performed. So, a data flow graph (DFG) can be constructed for each basic block. A node in this graph represents the operation performed by the associated instruction, and an arc from nodes k to l implies that the data produced by k is used by l. It then optimizes energy by sequentially performing four steps: simulated annealing based memory bank assignment, ASAP instruction packing, list scheduling, and finally, operand swapping if the basic block has filter operations. Tables IV and VI are looked up during list scheduling to reduce overhead cost, while Table XII is used for checking if operand swapping is beneficial.

VI. Experimental Results

Four DSP programs were tested to demonstrate the energy reductions possible by our minimization methodology. Table XIV shows the experimental results where all the comparisons are made in terms of the product of measured current and the number of cycles. These values are proportional to total energy. Multiplying them by 8.25×10^{-11}, gives the total energy in *Joules*. Column 1 lists the name of each benchmark program. The remaining columns show the energy comparisons by applying different minimization techniques: Column 2 (`org_un_p`) for the original unpacked code, Column 3 (`m`) for memory bank assignment alone, Column 4 (`m+p`) for the combined application of memory bank assignment and packing, Column 4 (`m+p+o`) for the combined application of memory bank assignment, packing, and overhead cost reduction by list scheduling, and Column 6 (`m+p+o+s`) for the combined application of all the techniques including operand swapping.

The first program `ex` is a real Fujitsu application for vector preprocessing. No **MAC** instructions are used in this program, so operand swapping is not applicable. The second program `LP_FIR60` is a length-60 linear phase FIR filter; the third program `IIR4` is a fourth-order direct form IIR filter; and the fourth program `FFT2` is a radix-2 decimal-in-time FFT butterfly. The last three programs are taken from the TMS320 embedded DSP examples in [11] and translated into native code for our target processor. In the case of `LP_FIR60`, because the same `MAC:LAB` instruction is repeatedly used, which is for filter operation on the data sequence, execution order does not change by different schedules. So list scheduling is not applicable to `LP_FIR60`. For each benchmark program, the product $I * N$ (which is proportional to energy) is given, where I is the measured average current in mA, and N is the number of execution cycles. The values in the parentheses are the reduction percentage compared with the products in column `org_un_p`. Thus, these values represent the energy reductions possible by the corresponding software modification technique. The percentages of the values in Table XIV relative to the products in column `org_un_p` (normalized to 1.00) are depicted in Fig. 13, which gives the relative energy percentages when different minimization steps are applied. Fig. 14(a) lists the original code for `IIR4`, and Fig. 14(b) lists the final energy optimized code.

The results show that about 9–39% energy reduction can be achieved by memory bank assignment alone. Then instruction packing can reduce energy by another 4–46%. The reason

```
*
* original org_un_p IIR4
*
* set pointer
        LDI     coefa,X0
        LDI     xn,X2
        MOV     (X2),C
        LDI     datanm1,X3
* d(n) = x(n)+d(n-1)a1+d(n-2)a2+
*              d(n-3)a3+d(n-4)a4
        MOV     A,(X3+1)
        MOV     B,(X0+1)
        MUL:
        MOV     A,(X3+1)
        MOV     B,(X0+1)
        MSMC:
        MOV     A,(X3+1)
        MOV     B,(X0+1)
        LDI     datan,X4
        MSMC:
        MOV     A,(X3+1)
        MOV     B,(X0+1)
        MSMC:
        LDI     coefb,x1
        MSMC:
        MOV     C,(X4)
* y(n) = d(n)b0+d(n-1)b1+d(n-2)b2+
*               d(n-3)b3+d(n-4)b4
        RESC:
        MOV     A,(X4+1)
        MOV     B,(X1+1)
        MUL:
        MOV     A,(X4+1)
        MOV     B,(X1+1)
        MSMC:
        MOV     A,(X4+1)
        MOV     B,(X1+1)
        MSMC:
        MOV     A,(X4+1)
        MOV     B,(X1+1)
        MSMC:
        MSMC:
* put y(n) in (X4)
        MOV     C,(X4)
*
```

(a)

```
*
* final IIR4 by m+p+o+s
*
* set pointer
        LDI     coefa,X0
        LDI     coefb,X1
        LDI     xn,X2
        LDI     datanm1,X3
        LDI     datan,X4
* d(n) = x(n)+d(n-1)a1+d(n-2)a2+
*              d(n-3)a3+d(n-4)a4
        MOV     (X2),C
        LAB     (X0+1),(X3+1)
        MUL:LAB (X0+1),(X3+1)
        MSMC:LAB (X0+1),(X3+1)
        MSMC:LAB (X0+1),(X3+1)
        MSMC:
        MSMC:
        MOV     C,(X4)
* y(n) = d(n)b0+d(n-1)b1+d(n-2)b2+
*               d(n-3)b3+d(n-4)b4
        RESC:LAB (X1+1),(X4+1)
        MUL:LAB (X1+1),(X4+1)
        MSMC:LAB (X1+1),(X4+1)
        MSMC:LAB (X1+1),(X4+1)
        MSMC:
        MSMC:
* put y(n) in (X4)
        MOV     C,(X4)
*
```

(b)

Fig. 14. (a) Original unpacked (`org_un_p`) `IIR4` code, (b) final low-energy optimized `IIR4` code (using `m+p+o+s`).

that `FFT2` has only 4% reduction is due to a certain data dependency in the unpacked code that prevents effective instruction packing.

After packing, the list scheduling algorithm for overhead cost reduction can further reduce energy by 4–14% for the packed codes. In the cases of `LP_FIR60`, `IIR4`, and `FFT2`, operand swapping is applicable, and an additional 18, 7, and 4% energy can still be saved, respectively. So, the overall energy reduction is seen to be 26–73% if the source code is originally unpacked and the memory banks are not assigned. Energy reductions of 8–17% are observed even if the original code is already preoptimized through the use of instruction packing and memory bank assignment.

VII. Conclusion and Future Work

In recent years, power/energy consumption has become one of the primary constraints in the design of embedded applications. In order to study the problem of energy consumption in these systems, one has to examine the variation in energy consumption due to variations in software, since software constitutes a large part of the functionality of embedded systems. This work describes the results of a study of the energy consumption due to software for a DSP processor that is used in several commercial applications. An empirical instruction-level power model has been developed for this processor. Due to architectural differences between this processor and others studied earlier, the power model has some important points of difference. These differences were exploited through a low-power instruction scheduling algorithm. Special architectural features that could be used for generating low energy code were identified and quantitatively analyzed, e.g., dual-memory accesses, instruction packing, and operand swapping for the Booth multiplier. Techniques for effectively utilizing these features were also described. Several examples were presented to illustrate the potential of all these techniques. Significant energy savings were observed. Energy savings obtained by these methods come for free—there is no hardware cost or software performance penalty. Future work will involve the refinement of the power model and optimization algorithms, and further exploration of code generation for low energy.

ACKNOWLEDGMENT

The authors would like to thank F. Hirose, H. Gambe, H. Fukui, T. Taniguchi, Y. Ohta, and N. Kumamoto of Fujitsu Ltd. for their assistance on experiment setup and technical discussion. The constructive comments by R. Brayton, R. Bryant, and A. Sangiovanni-Vincentelli are also highly appreciated.

REFERENCES

[1] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step toward software power minimization," in *IEEE Trans. VLSI Syst.*, pp. 437–445, Dec. 1994.
[2] V. Tiwari and Mike T.-C. Lee, "Power analysis of a 32-bit embedded microcontroller," *VLSI Design J.*, Accepted for publication, 1996.
[3] T. Sato, M. Nagamatsu, and H. Tago. "Power and performance simulator: ESP and its application for 100 MIPS/W class RISC design," in *Proc. Symp. Low Power Electron.*, San Diego, Oct. 1994, pp. 46–47.
[4] P.-W. Ong and R.-H. Yan, "Power-conscious software design—A framework for modeling software on hardware," in *Proc. Symp. Low Power Electron.*, San Diego, Oct. 1994, pp. 36–37.
[5] T. Sato, Y. Ootaguro, M. Nagamatsu, and H. Tago, "Evaluation of architecture-level power estimation for CMOS RISC processors," in *Proc. Symp. Low Power Electron.*, San Jose, CA, Oct. 1995, pp. 44–45.
[6] P. E. Landman and J. M. Rabaey, "Black-box capacitance models for architectural power analysis," in *Proc. Int. Workshop Low Power Design*, Napa, CA, Apr. 1994, pp. 165–170.
[7] M. T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power analysis and low-power scheduling techniques for embedded DSP software," in *Proc. 8th Int. Symp. Syst. Synthesis*, Cannes, France, Sept. 1995, pp. 110–115.
[8] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: An overview," in *Proc. Symp. Low Power Electron.*, San Diego, Oct. 1994, pp. 38–39.
[9] C.-L. Su, C.-Y. Tsui, and A. M. Despain, "Saving power in the control path of embedded processors," in *IEEE Design Test Comput.*, pp. 24–30, winter 1994.
[10] M. T.-C. Lee and V. Tiwari, "A memory allocation technique for low-energy embedded DSP software," in *Proc. Symp. Low Power Electron.*, San Jose, CA, Oct. 1995, pp. 24–25.
[11] Texas Instruments, *Digital Signal Processing Applications-Theory, Algorithm, and Implementations*, 1986.
[12] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," in *Science*, vol. 220, pp. 671–680, May 1983.
[13] A. Sudarsanam and S. Malik, "Memory bank and register allocation in software synthesis for ASIP's," in *Proc. Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 1995, pp. 388–392.
[14] M. Johnson, *Superscalar Microprocessor Design*. Englewood Cliffs, NJ: Prentice-Hall, Basic Software Scheduling, 1990, ch. 10.
[15] K. Hwang, *Computer Arithmetic—Principles, Architecture, and Design*. New York: Wiley, 1979.

Mike Tien-Chien Lee received the B.S. degree in computer science from National Taiwan University in 1987, and the M.S. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, in 1991 and 1993, respectively.

He was with David Sarnoff Research Center, Princeton, NJ, in 1993. Since 1994, he has been a Member of Research Staff at the Advanced CAD group of Fujitsu Laboratories of America, Santa Clara, CA. He was also a consulting researcher at Center of Reliable Computing, Stanford University, in 1994 and 1995. His research interests include high-level synthesis, low-power design, embedded system design, and test synthesis. He authored more than 20 techical papers in these areas, and a book titled *High-Level Test Synthesis of Digital VLSI Circuit* (Norwood, MA: Artech House, 1996).

Dr. Lee received Best Paper Awards at the Asia and South Pacific Design Automation Conference (ASP–DAC) in 1995 and the ACM/IEEE Design Automation Conference (DAC) in 1996. He is a member of the IEEE Computer Society and the ACM.

Vivek Tiwari received the B.Tech. degree in computer science and Engineering from the Indian Institute of Technology, New Delhi, India, in 1991. Currently, he is working toward the Ph.D. degree in the Department of Electrical Engineering, Princeton University, Princeton, NJ.

He joined Intel Corporation, Santa Clara, CA, in Fall 1996. His research interests are in the areas of computer aided design of VLSI and embedded systems and in microprocessor architecture. The focus of his current research is on tools and techniques for power estimation and low-power design. He has held summer positions at NEC Research Laboratories (1993), Intel Corporation in 1994, Fujitsu Laboratories of America in 1994, and IBM T. J. Watson Research Center in 1995, where he worked on the above topics.

Mr. Tiwari received the IBM Graduate Fellowship Award in 1993, 1994, and 1995, and a Best Paper Award at ASP–DAC'95.

Sharad Malik received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, New Delhi, India, in 1985 and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1987 and 1990, respectively.

Currently, he is an Associate Professor in the Department of Electrical Engineering, Princeton University, Princeton, NJ. His current research interests are in design tools for embedded computer systems, synthesis and verification of digital systems.

Dr. Malik has received the President of India's Gold Medal for Academic Excellence in 1985, the IBM Faculty Development Award in 1991, an NSF Research Initiation Award in 1992, Princeton University Rheinstein Faculty Award in 1994, the NSF Young Investigator Award in 1994, Best Paper Award at the IEEE International Conference on Computer Design in 1992 and at the ACM/IEEE Design Automation Conference in 1996, the Walter C. Johnson Prize for Teaching Excellence in 1993, and the Princeton University Engineering Council Excellence in Teaching Award in 1993, 1994, and 1995. He serves on the Program Committees of DAC, ICCAD, and ICCD. He is on the Editorial Boards of the *Journal of VLSI Signal Processing* and *Design Automation for Embedded Systems*.

Masahiro Fujita received the B.S. degree in electrical engineering in 1980, the M.S. degree in information engineering in 1982, and the Ph.D. degree in information engineering in 1985, all from the University of Tokyo, Tokyo, Japan. His Ph.D. dissertation Advisor was Prof. T. Motooka, Chairman of the Japanese Fifth Generation Computer Project.

From 1985 to 1993, he was employed as a Research Scientist by Fujitsu Laboratories in Kawasaki, Japan. Currently, he is director of Advanced CAD at Fujitsu Laboratories of America in Santa Clara, CA. He has been on program committees for many conferences dealing with digital design. His primary interest is in Computer-Aided Design for digital systems, especially logic synthesis and verification. He is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN AND INTEGRATED CIRCUITS AND SYSTEMS and *Formal Methods on Systems Design*.