6.3. SOFTWARE DESIGN FOR LOW POWER

KAUSHIK ROY AND MARK C. JOHNSON
School of Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana, U.S.A.

1. Introduction

It is tempting to suppose that only hardware dissipates power, not software. However, that would be analogous to postulating that only automobiles burn gasoline, not people. In microprocessor, micro-controller, and digital signal processor based systems, it is software that directs much of the activity of the hardware. Consequently, the software can have a substantial impact on the power dissipation of a system. Until recently, there were no efficient and accurate methods to estimate the overall effect of a software design on power dissipation. Without a power estimator there was no way to reliably optimize software to minimize power. Since 1993, a few researchers have begun to crack this problem. In this chapter, you will learn of the progress that has been made and identify ways to minimize the contribution of software to the power dissipation of mixed hardware-software designs.

We will start by exploring the sources of power dissipation that should be most influenced by software. The next section will look at how power dissipation can be modeled to facilitate power estimation of a particular application program. The core of this chapter will be a presentation of a variety of software power optimization techniques found in recent literature. The techniques will encompass three general categories: minimizing memory accesses, optimal selection and sequencing of machine instructions, and exploiting the low power features of some processors. Examples will be presented of design methodologies that incorporate these techniques. Differences between power and energy minimization will be discussed. Finally, we will examine several hardware design decisions that must take into account the intended software application in order to achieve minimum power dissipation. The chapter will close with a summary and a discussion of which techniques offer the greatest benefit in the design of a low power hardware-software system.

2. Sources of Software Power Dissipation

There are at least three contributers to CPU power dissipation that can be influenced significantly by software. The memory system takes of a substantial fraction of the power budget (on the order of one tenth to one fourth) for portable computers [10] and it can be the dominant source of power dissipation in some memory intensive DSP applications such as video processing [7]. System busses (address, instruction, and data) are a high capacitance component [24] for which switching activity is largely determined by the software. Datapaths in integer ALU and floating point units are demanding of power, especially if all operational units (e.g., adder, multiplier) or pipeline stages are activated at all times. Even power overhead for control logic and clock distribution are relevant since their contribution to a program's energy dissipation is proportional to the number of execution cycles of the program [27].

Memory accesses are expensive for several reasons. Reading or writing a memory location involves switching on highly capacitive data and address lines going to the memory, row and column decode logic, and word and data lines within the memory that have a high fanout. The mapping of data structures into multiple memory banks can influence the degree to which parallel loading of multiple words are possible [25]. Parallel loads not only improve performance, but are more energy efficient [15]. The memory access patterns of a program can greatly affect the cache performance of a system. Unfavorable access patterns (or a cache that is too small) will lead to cache misses and a lot of costly memory accesses. In multi-dimensional signal processing algorithms, the order and nesting of loops can alter memory size and bandwidth requirements by orders of magnitude [2]. Compact machine code decreases memory activity by reducing the number of instructions to be fetched and reducing the probability of cache misses [11]. Cache accesses are more energy efficient than main memory accesses. The cache is closer to the CPU than is main memory, resulting in shorter and less capacitive address and data lines. The cache is also much smaller than main memory, leading to smaller internal capacitance on word and data lines.

Busses in an instruction processing system typically have high load capacitances due to the number of modules connected to each bus and the length of the bus routes. The switching activity on these busses is determined to a large degree by software [24]. Switching on an instruction bus is determined by the sequence of instruction op-codes to be executed. Similarly, switching on an address bus is determined by the sequence and of data and instruction accesses. Both effects can be accounted at compile time. Data related switching is much harder to predict at compile time since most input data to a program is not provided until execution time.

Data paths for arithmetic logic units (ALU's) and floating point units (FPU's) make up a large portion of the logic power dissipation in a CPU. Even if the exact data input sequences to the ALU and FPU are hard to predict, the sequence of operations and data dependencies are determined during software design and compilation. The energy to evaluate an arithmetic expression might vary considerably with respect to the choice of instructions. A simple example is the common compiler optimization technique of reduction in strength where an integer multiplication by 2 could be replaced by a cheaper shift left operation. Poor scheduling of operations might result in unnecessary, energy wasting pipeline stalls. There are many other software design decisions that can affect data path power, but these will be discussed in section 4.

Some sources of power dissipation such as clock distribution and control logic overhead might not seem to have any bearing on software design decisions. However, each execution cycle of a program incurs an energy cost from such overhead. The longer a program requires to execute, the greater will be this energy cost. In fact, Tiwari et al. [27] found that the shortest code sequence was invariably the lowest energy code sequence for a variety of microprocessor and DSP devices. In no case was the lower average power dissipation of a slightly longer code sequence enough to overcome the overhead energy costs associated with the extra execution cycles. However, this situation may change as power management techniques improve, eliminating power consumption that does not contribute directly to the immediate computational task.

3. Software Power Estimation

The first step toward optimizing software for low power is to be able to estimate the power dissipation of a piece of code. This has been accomplished at two basic levels of abstraction. The lower level is to use existing gate level simulation and power estimation tools on a gate level description of an instruction processing system. A higher level approach is to estimate power based on the frequency of execution of each type of instruction or instruction sequence (i.e., the execution profile). The execution profile can be used a variety of ways. Architectural power estimation [22] determines which major components of a processor will be active during each execution cycle of a program. Power estimates for each active component are then taken from a look up table and added into the power estimate for the program. Another approach is based on the premise that the switching activity on busses (address, instruction, and data) is representative of switching activity (and power dissipation) in the entire processor. Bus switching activity can be estimated based on the sequence of instruction

op-codes and memory addresses [24]. The final approach we will consider is referred to as *instruction level power analysis* [16]. This approach requires that power costs associated with individual instructions and certain instruction sequences be characterized empirically for the target processor. These costs can be applied to the instruction execution sequence of a program to obtain an overall power estimate.

3.1. GATE LEVEL POWER ESTIMATION

Gate level power estimation [19, 5] of a processor running a program is the most accurate method available, assuming that a detailed gate level description including layout parasitics is available for the target processor. Such a detailed processor description is not likely to be available to a software developer, especially if the processor is not an in-house design. Even if the details are available, this approach will be orders of magnitude too slow for low power optimization of a program. The chief usefulness of a gate level power estimate is to evaluate the power dissipation behavior of a processor design and as a way to characterize the processor for the more efficient instruction level power estimation approaches.

3.2. ARCHITECTURE-LEVEL POWER ESTIMATION

Architecture-level power estimation is less precise but much faster than gate level estimation. This approach requires a model of the processor at the major component level (ALU, register file, etc.) along with power dissipation estimates for each component. It also requires a model of which components will be active as a function of the instructions being executed. The architecture-level approach is implemented in a power estimation simulator called ESP [22] (Early design Stage Power and performance simulator). ESP simulates the execution of a program, determining which system components are active in each execution cycle and adding the power contribution of each component.

3.3. BUS SWITCHING ACTIVITY

Bus switching activity is another indicator of software power based on a simplified model of processor power dissipation. In this simplified model, bus activity is assumed to be representative of the overall switching activity in a processor. Modeling bus switching activity requires knowledge of the bus architecture of a processor, op-codes for the instruction set, a representative set (or statistics) of input data to a program, and the manner in which code and data are mapped into the address space. By simulating the execution of a program, one can determine the sequence of op-codes, ad-

dresses, and data values appearing on the various busses. Switching statistics can be computed directly from these sequences of binary values. Su, Tsui, and Despain implement bus switching activity in their *cold scheduling* algorithm for instruction scheduling [24]. In this algorithm, only instruction and address related switching are considered due to the unpredictability of data values. However, for signal processing applications it is often possible to even predict switching activities due to correlations in sampled signal data [3].

3.4. INSTRUCTION-LEVEL POWER ANALYSIS

Instruction-level power analysis (ILPA) [27] defines an empirical method for characterizing the power dissipation of short instruction sequences and for using these results to estimate the power (or energy) dissipation of a program. A detailed description of the internal design of the target processor is not required. However, some understanding of the processor architecture is important in order to appropriately choose the types of instruction sequences to be characterized. This understanding becomes even more important when looking at ways to optimize a program based on the power estimate.

The first requirement for ILPA is to characterize the average power dissipation associated with each instruction or instruction sequence of interest. Three approaches have been documented for accomplishing this characterization. The most straightforward method, if the hardware configuration permits, is to directly measure the current draw of the processor in the target system as it executes various instruction sequences. If this is not practical, another method is to first use a hardware description language model (such as Verilog or VHDL) of the processor to simulate execution of the instruction sequences. An actual processor can then be placed in an automated IC tester and exercised using test vectors obtained from the simulation. An ammeter is used to measure current draw of the processor in the test system. A third approach is to use gate level power simulation of the processor to obtain instruction level estimates.

The choice of instruction sequences for characterization is critical to the success of this method. As a minimum, it is necessary to determine the *base cost* of individual instructions. Base cost refers to the portion of the power dissipation of an instruction that is independent of the prior state of the processor. Base cost excludes the effect of such things as pipeline stalls, cache misses, and bus switching due to the difference in op-codes for consecutive instructions. The base cost of an instruction can be determined by putting several instances of that instruction into an infinite loop. Average power supply current is measured while the loop executes. The loop should

be made as long as possible so as to minimize estimation error due to loop overhead (the jump statement at the end of the loop). However, the loop must not be made so large as to cause cache misses. Power and energy estimates for the instruction are calculated from the average current draw, the supply voltage, and the number of execution cycles per instruction.

Base costs for each instruction are not always adequate for a precise software power estimate. Additional instruction sequences are needed in order to take into account the effect of prior processor state on the power dissipation of each instruction. Pipeline stalls, buffer stalls, and cache misses are obvious energy consuming events whose occurrence depends on the prior state of the processor. Instruction sequences can be created that induce each of these events so that current measurements can be made. However, stalls and cache misses are effects that require a larger scale analysis or simulation of program execution in order to appropriately factor them into a program's energy estimate. There are other energy costs that can be directly attributed to localized processor state changes resulting from the execution of a pair of instructions. These costs are referred to as *circuit state effects*. Consider the following code sequence as an example:

$$ADD\ A \leftarrow B, C$$
$$MULT\ C \leftarrow A, B$$

The foremost circuit state effect is probably the energy cost associated with the change in the state of the instruction bus as the op-code switches from that for an addition operation to the op-code for a multiplication operation. Other circuit state effects for this example could include switching of control lines to disable addition and enable multiplication, mode changes within the ALU, and switching of data lines to reroute signals between the ALU and register file. Although this example examines a pair of adjacent instructions, it is also possible for circuit state effects to span an arbitrary number of execution cycles. This could happen if consecutive activations of a processor component are triggered by widely separated instructions. In such cases, the state of a the component may have been determined by the last instruction to use the component.

The circuit state cost associated with each possible pair of consecutive instructions is characterized for instruction level power analysis by measuring power supply current while executing an alternating sequence of the two instructions in an infinite loop. Unfortunately, it is not possible to separate the cost of an $A \rightarrow B$ transition from a $B \rightarrow A$ transition, since the current measurement is an average over many execution cycles.

Equation 1 from [27] concisely describes how the instruction level power measurements can be used to estimate the energy dissipation of a program.

$$E_P = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k E_k \quad (1)$$

E_P is the overall energy cost of a program, decomposed into base costs, circuit state overhead, and stalls and cache misses. The first summation represents base costs, where B_i is the base cost of an instruction of type i and N_i is the number of type i instructions in the execution profile of a program. The second summation represents circuit state effects where $O_{i,j}$ is the cost incurred when an instruction of type i is followed by an instruction of type j. $N_{i,j}$ is the number of occurrences where instruction type i is immediately followed by instruction type j. The last sum accounts for other effects such as stalls and cache misses. Each E_k represents the cost of one such effect found in the program execution profile.

TABLE 1. Instruction Set, Base Costs, and Circuit State Effects for MyDSP

Instruction Name	Base Cost [pJ]	Circuit State Effects [pJ]					
		LOAD	DLOAD	ADD	MULT	LOAD ADD	LOAD MULT
LOAD	1.98	0.13	0.15	1.19	0.92	1.25	1.06
DLOAD	2.37		0.17	1.19	0.92	1.32	1.06
ADD	0.99			0.26	0.53	0.86	0.99
MULT	1.19				0.66	0.79	0.96
LOAD;ADD	2.10					0.40	0.53
LOAD;MULT	2.25						0.79

A simple example will illustrate the estimation method. Let there be a hypothetical DSP called "MyDSP" whose instruction set includes those listed in table 1. MyDSP has registers named A, B, C, and D. Each instruction on MyDSP executes in one cycle which is 25ns. Assume that someone has measured the currents for each instruction and instruction pair in the manner described earlier. The currents were then used to calculate the base energy cost for each instruction and circuit state effect energy for each possible instruction pair. These costs are shown in table 1. In reality, these numbers were contrived to resemble a Fujitsu DSP documented in [27].

The numbers will be used later to illustrate the effect of some code optimizations. Let us analyze the energy and average power associated with the following code fragment running on MyDSP that evaluates the expression $(x \times y) + z$, where x, y, and z are memory variables. We will assume that the code fragment does not cause any stalls or cache misses.

$DLOAD\ A \leftarrow x, B \leftarrow y$	# prior instruction was ADD
$LOAD\ C \leftarrow z; MULT\ D \leftarrow A, B$	# LOAD & MULT packed together
$ADD\ A \leftarrow C, D$	# A now contains $(x \times y) + z$

The energy and power estimate can be tabulated as follows:

Instruction Name	Base Cost [pJ]	Circuit State [pJ]	
DLOAD	2.37	1.19	
LOAD;MULT	2.25	1.06	
ADD	0.99	0.99	
Totals:	5.61	3.24	Total Energy = 8.85pJ Ave. Power = 8.85pJ/75ns = 118uW

The base cost for each instruction is taken from the second column of table 1. We have to look at adjacent pairs of instructions in order to determine the circuit state costs. We will assume that the DLOAD instruction was preceded by an ADD. The circuit state cost for the DLOAD can then be found at the intersection of the row labeled "DLOAD" and the column labeled "ADD". Similarly, the intersection of row "DLOAD" and column "LOAD;MULT" gives the circuit state cost for the LOAD;MULT. The base and circuit state costs are all added together to obtain the energy cost of the code sequence. Energy is divided by execution time to obtain average power.

The previous example demonstrates the concept of ILPA for a trivial code fragment. The procedure is automated and integrated into a process for analyzing the power dissipation of real-world applications. Basic blocks are first extracted from the assembly or object code for a program. Instruction level base costs and circuit state effects are totaled for each block. The frequency and cost of stalls for each basic block are estimated and added into the basic block cost. A program profiler determines how many times each basic block is executed in order to accumulate the total basic block costs for the program. Finally, a cache simulator is used to estimate the

frequency of cache misses so that cache miss costs can be added to the total program energy.

4. Software Power Optimizations

Software optimizations for minimum power or energy tend to fall into one or more of the following categories: selection of the least expensive instructions or instruction sequences, minimizing the frequency or cost of memory accesses, and exploiting power minimization features of hardware. The following subsections will discuss each of these categories, present several optimization techniques that have been documented, and relate the techniques back to the sources of software power dissipation identified in section 2.

A prerequisite to optimizing a program for low power must always be to design an algorithm that maps well to available hardware and is efficient for the problem at hand in terms of both time and storage complexity. Given such an algorithm, the next requirement is to write code that maximizes performance, exploiting the performance features and parallel processing capacity of the hardware as much as possible. In doing so, we will have gone a long way to minimize the energy consumption of the program [27]. Maximizing performance also gives increased latitude to apply other power optimization techniques such as reduced supply voltages, reduced clock rates, and shutting off idle components.

It should be clarified that the energy optimization objectives differ with respect to the intended application. In battery powered systems, the total energy dissipation of a processing task is what determines how quickly the battery is spent. In systems where the power constraint is determined by heat dissipation and reliability considerations, instantaneous or average power dissipation will be an important optimization objective or constraint. An instantaneous power dissipation constraint could lead to situations where one would need to deliberately degrade system performance through software or hardware techniques. A hardware technique is preferable since hardware performance degradation can be achieved through energy saving techniques (lower voltage, lower clock rate). Software performance degradation is likely to increase energy consumption by increasing the number of execution cycles of a program.

4.1. INSTRUCTION SELECTION AND ORDERING

In section 3, we looked at various ways that the power and energy of a program can be estimated from assembly or machine code. Since there are usually many possible code sequences that accomplish the same task, it should be possible to select a code sequence that minimizes power or energy. In fact, many code optimization techniques for speed or code size should

be equally applicable to power or energy optimization; the main difference is the objective function used in code generation decisions. We will not attempt to give a comprehensive presentation of general code optimization techniques. Such techniques are well documented in compiler literature. Instead, we will attempt to provide a basis for judging how different kinds of coding techniques are likely to affect energy usage. We will also discuss some specific techniques that researchers have evaluated for their energy savings: *instruction packing*, *minimizing circuit state effects*, and *operand swapping*. These techniques are described in greater detail in [16].

Given the benefits of high performance code with respect to energy savings, optimizations that improve speed should be especially helpful. Code size minimization may be necessary, especially in embedded systems, but it is not quite as directly linked to energy savings. Regarding performance and energy minimization, cache performance is a greater concern. Large code and a small cache can lead to frequent cache misses with a high power penalty. Code size and cache performance are concerns that motivate the effort to maximize code density for embedded processors [11].

Instruction packing is a feature of a Fujitsu DSP [27] and a number of other processors. Instruction packing permits an ALU operation and a memory data transfer to be packed into a single instruction. Much of the cost of each individual instruction is overhead that is not duplicated when operations are executed in parallel. Consequently, there can be nearly a 50% reduction in energy associated with the packing of two instructions. Similar benefits can be achieved by parallel loading of multiple words from different banks of memory. Optimization of memory accesses is a broad topic that is covered in section 4.2, including an example that illustrates the benefit of parallel loads and instruction packing. Concurrent execution of integer and floating point operations represent still another opportunity to minimize the total energy of a group of operations. All of these techniques are really special cases of the types of coding that can be done in VLIW (very long instruction word) and superscalar architectures where opportunities for instruction packing will be much more extensive.

Instruction ordering for low power attempts to minimize the energy associated with the circuit state effect. Circuit state effect, if you recall, is the energy dissipated as a result of the processor switching from execution of one type of instruction to another. On some types of processors, especially DSP's, the magnitude of the circuit state effect can vary substantially depending on the pair of instructions involved. In section 3.4, we described techniques for measuring or estimating the circuit state effect for different pairs of consecutive instructions. Given a table of circuit state costs, one can tabulate the total circuit state cost for a sequence of instructions. Instruction ordering then involves reordering instructions to minimize the

total circuit state cost without altering program behavior. Researchers have found that in no case do circuit state effects outweigh the benefit of minimizing program execution cycles [27]. Circuit state effects are found to be much more significant for DSP's than for general purpose architectures.

Accumulator spilling and mode switching are two DSP behaviors that are sensitive to instruction ordering [17] and are likely to have some power impact. In some DSP architectures, the accumulator is the only general purpose register. With a single accumulator, any time an operation writes to a new program variable, the previous accumulator value will need to be spilled to memory, incurring the energy cost of a memory write. In some DSP's, operations are modal. Depending on the mode setting (such as the sign extension mode), datapath operations will be handled differently. We have not seen mode switching evaluated with respect to the energy costs, but a mode switch will have execution time and capacitive switching costs associated with it.

Operand swapping attempts to swap operands to ALU or floating point operations in a way that minimizes the switching activity associated with the operation. One way operand swapping could help is to minimize switching related to the sequence of input values. If latches are located at the inputs of a functional unit to keep the inputs from switching during idle periods, then there could be an advantage to swapping operands to minimize switching at the inputs to the functional unit. For example, if two consecutive additions have one operand that stays the same (say $x + 7$ and $y + 7$), then switching would be minimized if the common operand was applied to the same adder input both times.

Ordering of operands can have an even greater effect if the operands to a commutative operation are not treated symmetrically. For example, the number of additions and subtractions performed in a Booth multiplier depends on the bit pattern of the second operand. The number of additions and subtractions required is referred to as the *recoding weight* of the second operand. Given a pair of values to be multiplied, power should be reduced if we put the value with the lower recoding weight into the second input of the Booth multiplier. We rarely know ahead of time the value of both operands, but it is not unusual to know the value (and recoding weight) of one operand. Following is a simple and effective heuristic to apply this knowledge. If an operand is known to have a high recoding weight, put it in the first operand position. If an operand is is known to have a low recoding weight, put it in the second position. Lee [16] did this for several sets of operand values and demonstrated that in a majority of cases, a substantial energy savings per operation on the order of 10% to 30% was realized. For the few cases when swapping was not beneficial, the penalty was modest, less than 6%.

Since memory often represents a large fraction of a system's power budget, there is a clear motivation to minimize the memory power cost. Happily, software design techniques that have been used to improve memory system efficiency can also be helpful in reducing the power or energy dissipation of a program. This is largely due to memory being both a power and a performance bottleneck. If memory accesses are both slow and power hungry, then software techniques that minimize memory accesses tend to have both a performance and a power benefit. Lower memory requirements also help by allowing total RAM to be smaller (leading to lower capacitance) and improving the percentage of operations that only need to access registers or cache. Power minimization techniques related to memory concentrate on one or more of the following objectives:

- Minimize the number of memory accesses required by an algorithm.
- Minimize the total memory required by an algorithm.
- Make memory accesses as close as possible to the processor; registers first, cache next, external RAM last.
- Make the most efficient use of the available memory bandwidth; e.g., use multiple word parallel loads instead of single word loads as much as possible.

For multi-dimensional signal processing applications, the nesting of loops controlling array operations and the order of operations can substantially influence the number of memory transfers and the total storage requirements. Catthoor et al. [2] presented three simple examples, representative of some typical signal processing loop structures, that illustrate the impact of loop nesting and operation ordering. These examples are reproduced below along with brief discussions. In each case, M and N are large integer values.

Example 1:

Before:	After:
FOR i:= 1 TO N DO	FOR i := 1 TO N DO BEGIN
B[i] := f(A[i]);	B[i] := f(A[i]);
FOR i := 1 TO N DO	C[i] := g(B[i]);
C[i] = g(B[i]);	END;

In example 1, the B array is too large to store in registers, so memory transfers are required to store and retrieve the intermediate values stored in B. Re-arranging the loops allows the intermediate values to be kept in

a register. $2N$ memory transfers are replaced by cheaper register accesses and N memory locations are no longer needed.

Example 2:

Before:	After:
FOR i:= 1 TO N DO	FOR j := 1 TO N DO
B[i] := f(A[i]);	D := g(C[j],D);
FOR j := 1 TO N DO	FOR i := 1 TO N DO
D := g(C[j],D);	B[i] := f(A[i]);

Example 3:

Before:	After:
FOR j := 1 TO M DO	FOR i := 1 TO N DO BEGIN
FOR i := 1 TO N DO	FOR j := 1 TO M DO
A[i][j] := g(A[i][j-1],D);	A[i][j] := g(A[i][j-1],D);
FOR i := 1 TO N DO	OUT[i] := A[i][M];
OUT[i] := A[i][M];	END;

The transformations shown in examples 2 and 3 do not improve on the number of memory transfers, but space requirements are reduced. In example 2 if C is not needed afterwards, operations are reorganized so that B can be mapped into the same storage as C, eliminating N storage locations. In example 3, storage for intermediate values $A[i][M]$ can be reduced from N locations to 1 by restructuring and combining loops.

An example where careful mapping of an algorithm to hardware was beneficial in terms of memory performance, size requirements, and power was accomplished by DeGreef [7]. DeGreef evaluated the mapping of a video motion estimation algorithm onto a variety of digital signal processors. The algorithm requires enormous memory bandwidth since 5.3G pixels must be processed per second to support real time processing of a standard television signal. Use of cache was essential, but cache protocols such as a least-recently-used replacement policy can lead to unacceptable variations in execution time for such a time critical application. Many signal processing algorithms such as motion estimation can have very predictable control flow and memory access patterns. Consequently, it may be possible to derive an optimal cache policy. DeGreef used the on-chip memory of a DSP as a data cache for which storage and replacement decisions could be made at compile time. With a cache as small as 23Kbytes, it was possible to reduce accesses

to external memory by a factor of more than 100 as compared to no cache at all. Another technique applied by DeGreef was to minimize loop control overhead and insert dummy code to synchronize the DSP software to the video I/O data rate. This allowed I/O, current, and previous video frame buffers to be merged into one frame buffer. Synchronization was already known as a useful tool to reduce buffers, but there should also be a power benefit resulting from a smaller memory and simpler I/O management.

Another recent example of a beneficial mapping of an algorithm to an architecture was presented by Lidsky [18]. The application was vector quantization of video data. Each 4x4 pixel region of an image had to be quantized to the nearest vector in a 256 element codebook of pre-computed vectors. Decoding a quantized image requires a simple table lookup. However, encoding a 4x4 pixel region requires a search of the entire codebook. Lidsky organized the codebook into a binary search tree as illustrated in figure 1. The number shown inside each node indicates a level in the tree. Starting at the top node, a vector to be quantized is compared to two codebook entries in the next level down. The closer node determines the branch that is followed in the search tree. The tree below the farther node is eliminated from the search. The process is repeated until a leaf of the tree is reached. This kind of search requires $order(log_N)$ operations to search the tree. A full linear search would have required $order(N)$ operations.

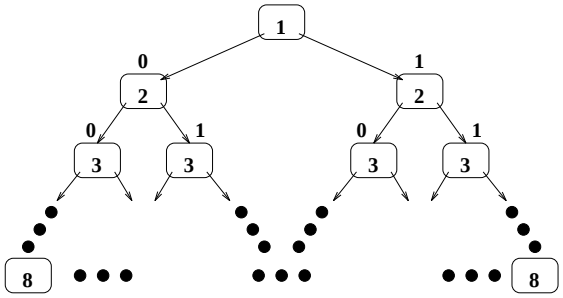

Figure 1. Binary codebook search.

The binary search algorithm is not only much more efficient than linear search, it also permits a fast pipelined implementation. A separate memory and processing element can be used for each level of the search tree. The encoding process can then proceed in a pipelined manner, handling eight vectors simultaneously. The clock rate can be reduced by a factor of eight from a serial implementation. This architecture increases memory area more than twofold, but short critical paths and reduced clock permit a supply voltage reduction from 3 to 1.1 volts. Overall, memory accesses

were reduced by a factor of 30 and power was scaled down by a factor of 17 compared to a system with a single serial access memory.

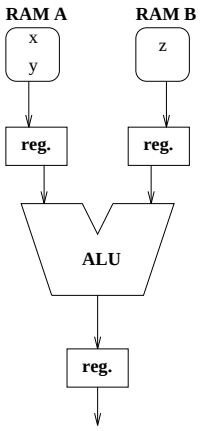

Figure 2. Datapath example for dual memory load.

TABLE 2. Effect of Memory Mapping on Energy and Power of MyDSP

	(a)	(b)	(c)
RAM A	x,y,z	x,y	x,z
RAM B		z	y
	LOAD B←x	DLOAD B←x,A←z	DLOAD B←x,C←y
	LOAD C←y	LOAD C← y	LOAD A←z; MULT B ← B,C
	LOAD A←z; MULT B← B,C	MULT B←B,C	ADD A←A,B
	ADD A←A,B	ADD A←A,B	
Energy [pJ]	10.57	9.32	8.85
Power [uW]	105.7	93.2	118.0

In general purpose systems it may not be practical to craft the software and hardware so finely together, but there are techniques which are more readily applicable to a wide variety of problems. One of these techniques is to maximize parallel loads of multiple words from memory. Dual loads are beneficial for energy reduction, but not necessarily for power reduction because the instantaneous power dissipation of a dual load is somewhat higher

than for two single loads. Tiwari et al. [27] demonstrated energy savings of as much as 47% by maximizing dual loads. Achieving this savings required two phases. The first phase optimized the memory allocation of a program to maximize opportunities for using dual loads. Memory accesses were then combined as much as possible. Suppose that the datapath depicted in figure 2 [15] is required to evaluate the expression, $(x \times y) + z$ as in the example from section 3.4. Suppose also that there are two memory banks, all operations must be performed on register values, and the processor supports dual loads and packing together of ALU and memory operations. Depending on the memory allocation and objective chosen, the optimal instruction sequence choice can vary. Table 2 uses the the hypothetical characterization of MyDSP (table 1) to illustrate what could happen. In case (a) all variables are in the same memory block, so no dual loads were possible. It was possible to overlap a load and a multiply. In (b), x and z could be loaded in parallel but no other operations could be overlapped. In (c), a dual load of x and y and parallel execution of a load and multiply were both possible. The lowest energy solution occurred when the memory allocation permitted maximum parallelism of operations. Lower average power solutions were obtained at the expense of execution cycles and total energy.

The previous example illustrates the energy benefit of favorable memory allocation, but algorithms are needed to obtain this benefit for entire programs, not just code fragments. Researchers have dealt previously with the problem of automated memory allocation for optimal performance, examples include [6] and [25]. Although performance was their target, reduced program execution times translate directly to reduced program energy consumption and can also provide opportunities for voltage and clock frequency reduction. Lee and Tiwari [15] have presented a memory allocation technique targeted specifically for energy reduction on systems with partitioned memory. Davidson [6] listed several practices that minimize memory bandwidth requirements:

- Register allocation to minimize external memory references.
- Cache blocking, i.e., transformations on array computations so that blocks of array elements only have to be read into cache once.
- Register blocking, similar to cache blocking except that redundant register loading of array elements is eliminated.
- Recurrence detection and optimization, i.e., use registers for values that are carried over from one level of recursion to the next.
- Compact multiple memory references into a single reference.

All of the researchers just cited focus on combining memory references, but they take substantially different approaches. Combining of memory references is of special interest in part because it is an objective that was

not well addressed by earlier compiler technology. We will briefly review the methods applied by Davidson, Sudarsanam, and Lee.

Davidson's approach [6] evaluates the benefit of *loop unrolling* on overall performance and on the degree to which *memory reference compaction* (also called *memory access coalescing*) opportunities are uncovered. Loop unrolling transforms several iterations of a loop into a block of straight-line code. The straight-line code is then iterated fewer times to accomplish the same work as the original loop. The purpose of the transformation is to make it easier to analyze data dependencies when determining where to combine memory references. Davidson found that the performance benefit of loop unrolling and memory access coalescing varied a great deal depending on the processor. Speedups as much as 40% for a DEC alpha and 25% for a Motorola 88100 were observed. On the other hand, performance was reduced on a Motorola 68030.

Sudarsanam [25] combines register and memory bank allocation into a unified optimization phase after instruction selection. First, all straight-line blocks of code in a program are compacted, combining two memory operations or a memory and an ALU operation as much as possible without changing the order of operations. A *constraint graph* is then generated that represents any register or memory allocation constraints that are implied by the compacted code. In the constraint graph, each vertex represents a symbolic register or variable. Each edge represents an allocation constraint on the pair of vertices joined by the edge.

Several types of edges are defined and color coded to represent different types of constraints. Figure 3 provides examples of some of these edge types. The label inside the circle for each vertex indicates a variable (e.g., var_v), a symbolic register name (e.g., reg_v), or a physical register name (e.g., X). The label just outside of each circle indicates the assignment of a physical register name to a symbolic register or a memory bank name to a variable. A red edge indicates that two register values are active at the same time and should be assigned to different physical registers. A green edge identifies a parallel memory to register transfer that should be preserved. Brown, blue, and yellow edges all indicate other types of parallel data transfers such as a memory access in parallel with a register-register transfer. The brown, blue, and yellow edges have a similar structure to the green edge. The vertices representing two destination registers or variables are connected by an edge. At each end of the edge, a pointer is attached to the source register or variable. Black edges indicate limitations on operand usage imposed by the target processor (perhaps only certain registers can be used as input to ALU operations). A black edge is connected at one end to a symbolic register name. The other end is connected to a physical register name. Each black edge identifies an impossible register assignment.

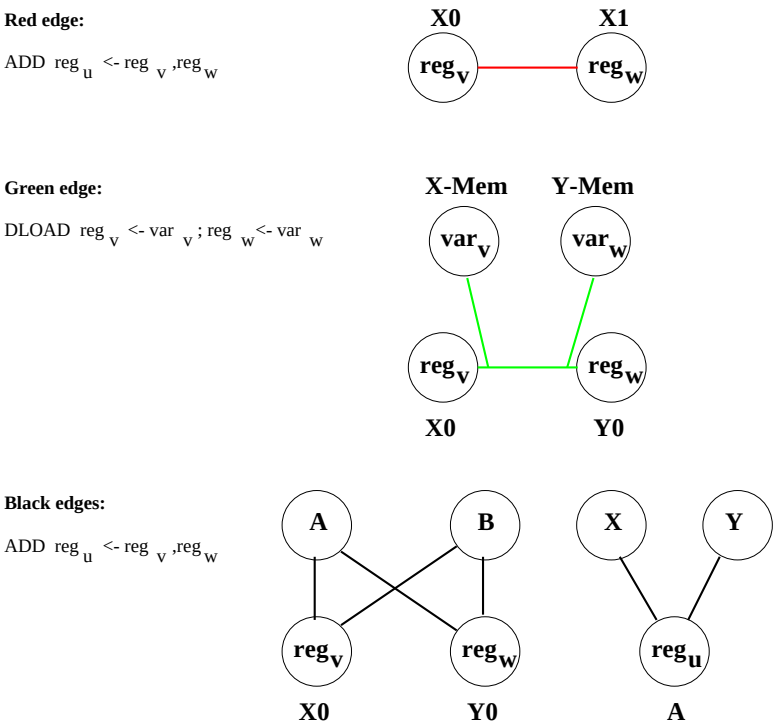

Figure 3. Sample memory and register allocation constraints.

Every edge has a weight (not shown in figure 3) that indicates the penalty of violating the constraint. The register and memory bank allocation is then accomplished by using a simulated annealing algorithm to label each vertex.

Lee and Tiwari [15] present an algorithm for memory bank assignment that is formulated as a graph partitioning problem and solved by simulated annealing. Each vertex represents a variable. Each edge indicates that there exists an ALU operation that requires a particular pair of variables as arguments. Consider the following code fragment

$$
\begin{array}{llll}
\text{ADD} & R_1 & \leftarrow & \text{a,e} \\
\text{ADD} & R_2 & \leftarrow & \text{e,b} \\
\text{ADD} & R_3 & \leftarrow & \text{a,b} \\
\text{ADD} & R_4 & \leftarrow & \text{b,d} \\
\text{ADD} & R_5 & \leftarrow & \text{a,c} \\
\text{ADD} & R_6 & \leftarrow & \text{c,d}
\end{array}
$$

The access graph for this code could be drawn as shown in figure 4.

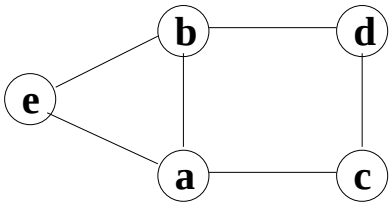

Figure 4. Access graph for code fragment.

A two-way partition of this graph directly corresponds to a memory bank assignment. All of the variables (vertices) in one partition are assigned to the same memory bank. The cost of a partition is evaluated in terms of the number of execution cycles required to accomplish all of the memory transfers indicated by the graph. If an edge lies entirely in one side of the partition, two memory transfers will be required for the indicated variables. If an edge crosses the partition, then the memory accesses for those two variables can be compacted into a dual load operation at a cost of one execution cycle. The partition shown in figure 5 maximizes the number of edges crossing the partition. The cost of this partition is 7: 2 for the a, e edge and 1 for all other edges. Clearly this cost is an upper bound on the optimum number of loads since appropriate allocation of registers and sequencing of instructions could reduce the cost to two parallel loads and one single word load.

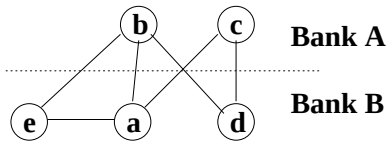

Figure 5. Partitioned access graph.

Cache locking, if the processor supports it, provides another way to minimize memory accesses. Locking prevents any memory reads and writes from going to main memory. The real benefit comes from cache write hits not being written through to main memory. For read hits there would be no reference to main memory even if the cache were unlocked. The cost of memory writes have been measured for both locked and unlocked caches [26]. For a Fujitsu SPARClite MB86934 32 bit RISC processor, writing a value of 0 drew 341mA from the power supply when the cache was unlocked, but just 194mA with the cache locked. Similar results have been reported for a variety of write operations.

Most of the low power software optimizations that we have discussed already exploit specific hardware features to achieve maximum benefit. Many of these features are not unique to low power designs. However, there are some current techniques and ideas for future low power designs that will require consideration in the software design process. Such low power techniques include software control of power management, multi-processor low power architectures, and voltage/clock rate scaling.

Several processors support varying levels of software control over power management. Some examples include the Fujitsu SPARClite MB86934 [26], Hitachi SH3 [11], and Intel486SL [8]. The SPARClite has a *power-down* register that masks or enables clock distribution to the SDRAM interface, DMA module, floating-point unit, and floating point queues. The Hitachi SH3 provides two power reduction modes, *standby* and *sleep*, along with control over the CPU and peripheral clock frequencies. In sleep mode, all operations except the real time clock stop. In standby mode, the CPU core is stopped, but the peripheral controller, bus controller, and memory refresh operations are maintained. The Intel486SL provides a System Management Mode (SMM), a distinct operating mode that is transparent to the operating system and user applications. The SMM is entered by means of an asynchronous interrupt. Software for the SMM is able to enable, disable, and switch between fast and slow clocks for the CPU and ISA bus.

A trade-off must be made regarding the merits of software control vs. automated hardware control of power management. Software based power management has the advantage of better information upon which to base power management decisions. The software designer and compiler are in a better position to determine when removing or slowing down a clock will be of benefit. Purely hardware based power management has to make its decisions by monitoring processor activity. An incorrect decision to power down can hurt performance and power dissipation due to the cost of restoring power and system state [23]. Processors with integrated power management provide ways to efficiently save and restore processor state in the event of a power-down. Program execution cycles committed to power management represent the down side of software based power management.

Parallel processors offer a means to use increased hardware to obtain reduced supply voltage and power dissipation. Srivastava [23] provided an analysis demonstrating that slow, parallel hardware operating at a low supply voltage should be more efficient than a single processor operating at a higher voltage, given applications with adequate algorithm concurrency. Parallel processor compiler technology must be brought to bear in order to take advantage of this trade-off.

Given a processor with dynamic control of both supply voltage and clock speed, Yao et al. [28] offer job level scheduling algorithms that minimize CPU energy. Their approach takes as input a set of jobs and time windows for completion of each job. A schedule is generated that specifies job start times, completion times, and processor speed vs. time. An optimal algorithm is offered for static scheduling and a heuristic for on-line scheduling.

5. Automated Low Power Code Generation

Section 4 described a variety of optimizations and software design approaches that can minimize energy consumption. To be used extensively, those techniques need to be incorporated into automated software development tools. There are two levels of tools: tools that generate code from an algorithm specification and compiler level optimizations.

We have not located any comprehensive high level tools intended for low power code development, but several technologies appear to be available to build such tools, especially for DSP applications. Graphical [14] and textual languages [12] have been available for some time that enable one to specify a DSP algorithm in a way that doesn't obscure the natural parallelism and data flow. HYPER-LP [4] is a DSP datapath synthesis system that incorporates several algorithm level transformations to uncover parallelism and minimize critical paths so that the datapath supply voltage can be minimized. Even though HYPER-LP is targeted for datapath circuit synthesis, the same types of transformations should be useful in adapting an algorithm to exploit parallel resources in a DSP processor or multi-processor system (in fact HYPER has been used to evaluate algorithm choices prior to choice of implementation platform [20]). MASAI [2] is a tool that reorganizes loops in order to minimize memory transfers and size. Finally, to complete the tool set, there exist compilers including [9, 24, 16] that either already optimize DSP code for low power or could be enhanced to apply the techniques that have been discussed.

Compiler technology for low power appears to be further along than the high level tools, in part because well understood performance optimizations are adaptable to energy minimization. This is true more for general purpose processors than for DSP processors. DSP compiler technology faces difficulties in dealing with a small register set (possibly just an accumulator), irregular data paths, and making full use of parallel resources. The rest of this section describes two examples of power reduction techniques that have been incorporated into compilers.

Su [24] presented a *Cold Scheduling Algorithm* which is an instruction scheduling algorithm that reduces bus switching activity related to the

change in state when execution moves from one instruction type to another. The algorithm is a list scheduler that prioritizes the selection of each instruction based on the power cost (bus switching activity) of placing that instruction next into the schedule. Following is the ordering of compilation phases that Su proposed in order to incorporate cold scheduling.

1. Allocate Registers.
2. Pre-assemble: Calculate target addresses, index symbol table, transform instructions to binary.
3. Schedule instructions using cold scheduling algorithm.
4. Post-assemble: Complete the assembly process.

The assembly process was split into two phases to accommodate the cold scheduler. Cold scheduling could not be performed prior to pre-assembly because the binary codes for instructions are needed to determine the effect of an instruction on switching activity. Scheduling could not be the last phase since completion of the assembly process requires an ordering of instructions. Su pointed out that one limitation of this approach is that it becomes difficult to schedule instructions across basic block boundaries because of the early determination of target addresses. Cold scheduling was found to obtain a 20% to 30% switching activity reduction with a performance loss of 2% to 4%. It is easy to imagine that the instruction level power model (section 3.4) could also be used to prioritize instruction selection during cold scheduling.

Lee et al., proposed a code generation and optimization methodology that encompasses several of the techniques discussed in section 4. Following is the sequence of phases that they proposed:

1. Allocate registers and select instructions.
2. Build a data flow graph (DFG) for each basic block.
3. Optimize memory bank assignments by simulated annealing.
4. Perform *as soon as possible* (ASAP) packing of instructions.
5. Perform list scheduling of instructions (similar to cold scheduling)
6. Swap instruction operands where beneficial.

The optimization phases appear to be sequenced in order from greatest to least incremental benefit. Overall energy savings on four benchmarks ranged from 26% to 73% compared to results that did not incorporate instruction packing nor optimized memory bank assignments.

6. Codesign for Low-Power

The emphasis of this chapter has been on software design techniques for low power. However, it is probably obvious by now that many of these techniques are dependent in some way upon the hardware in order for a power

savings to be realized. Hardware/Software codesign for low power is a more formal term for this problem of crafting a mixture of hardware and software that provides required functionality, minimizes power consumption, and satisfies objectives that could include latency, throughput, and area. Instruction set design and implementation seems to be one of the most well defined of the codesign problems. In this section, we will first look at instruction set design techniques and their relationship to low power design. We will then survey the hardware/software trade-offs that have come up in our consideration of power optimization of software.

6.1. INSTRUCTION SET DESIGN

Some of the most recent efforts at instruction set optimization were presented by Sato et al. [21], Binh et al. [1], and Huang and Despain [13]. The work of Sato and Binh are both related to the PEAS-I system (Practical Environment for ASIP development-version I). PEAS-I is a system that synthesizes an HDL (hardware description language) description of a CPU core along with a C compiler, assembler, and simulator for that CPU. Inputs to PEAS-I are a set of design constraints (including chip area and power consumption), a hardware module database, a sample application program and program dataset. PEAS-I optimizes the instruction set and instruction implementations for the application program and dataset that was given. PEAS-I defines an extended set of instructions by starting with a core instruction set that would be needed to implement any C program. The core set is augmented with instructions corresponding to any C operators not already represented by a single instruction. The set could be further expanded to include any pre-defined or user defined functions. For each possible instruction, alternative implementations are defined including hardware, microprogram, and software options. Optimal selection of implementation methods for each instruction is defined as an integer program and solved by a branch and bound algorithm. The power and area contribution of each instruction implementation is estimated and added together to include in power and area constraints. The number of execution cycles of each instruction is multiplied by the frequency of occurrence in the sample application and added together to form the objective function for the optimization. Binh's contribution was to extend the instruction selection formulation to account for pipeline hazards.

Huang and Despain's approach is similar in one respect to PEAS-I, in that it optimizes the instruction set for sample applications. The most significant difference is that Huang's approach groups *micro-operations* (MOPS) together to form higher level instructions. Each benchmark is expressed as a set of MOPS with dependencies to be satisfied. MOPS are

merged together as a byproduct of the scheduling process. MOPS that are scheduled to the same clock cycle are combined. For any candidate schedule and instruction set, instruction width (bits), instruction set size, and hardware resource requirements are constrained. The scheduling problem

TABLE 3. Memory System Considerations for Low Power Software

Memory System Feature	Low-Power Software Impact
Total memory size	Minimizing total memory requirements through code compaction and algorithm transformations allow for a smaller memory system with lower capacitances and improved performance. However, this benefit might have to be traded off against a faster algorithm that requires more memory.
Partitioning into banks How many, how big	Needed if parallel loads are to be used. The size of application, data structures, access patterns should determine this.
Wide data bus	Needed for parallel loads.
Proximity to CPU	Memory close to CPU reduces capacitances, makes memory use less expensive.
Cache size	Should be sized to minimize cache misses for the intended application. Software should apply techniques such as cache blocking to maintain high degree of spatial and temporal locality.
Cache protocol	If an application's memory access patterns are well defined, the protocol can be optimized.
Cache locking	If significant portions of an application can run entirely from cache, this can be used to prevent any external memory accesses while those portions of code are executing.

is solved by simulated annealing with an objective of minimizing execution cycles and instruction set size.

It would seem that any of the power estimation techniques described in section 3 could fit nicely into PEAS-I or Huang's approach. Either approach relies on instruction level application profiles to prioritize decisions regarding the instruction set. Conveniently, the architecture-level, bus switching, and instruction-level power estimates can decompose power estimates at the instruction level. The challenging task would be determining appropriate instruction level power estimates and circuit state effects for an unsynthesized CPU. Op-code selection is another design decision that could be optimized for low power, considering the impact of the op-codes on circuit state effects.

TABLE 4. Architectural Considerations for Low Power Software

Architectural Feature	Low-Power Software Impact
Class of processor DSP/RISC/CISC	Application dependent. How well does the instruction set fit the application? Is there hardware support for operations common to the application? Does the hardware implement functions not needed by the application?
Parallel processing VLIW, Superscalar SIMD, MIMD How much parallelism?	Does the level and type of parallelism in an algorithm fit one of these architectures? If so, the parallel architecture can greatly improve performance and allow reduced voltages and clock rates to be used.
Bus architecture	Separate busses (e.g., address, data, instruction, I/O) can make it easier to optimize instruction sequences, addressing patterns, and data correlations to minimize bus switching.
Register file size	Increased register count eases register allocation and reduces memory accesses, but too large of a register file can make register accesses as expensive as a cache access.

6.2. ARCHITECTURE AND CIRCUIT LEVEL DECISIONS

Software power reduction is influenced by a number of hardware decisions, particularly in the areas of memory system design, processor architecture, and hardware support for power management. Tables 3 through 5 list hardware features that should have the most impact on how software is optimized. For each feature, the interrelationship with software and the intended application is discussed.

TABLE 5. Power Management Considerations for Low Power software

Power Management Feature	Low-Power Software Impact
Software vs. Hardware control	Software control allows power management to be tailored to the application, at the cost of added execution cycles.
Clock/voltage removal At what level of granularity?	Some kind of shutdown mode is needed in order to realize a benefit from minimized execution times.
Clock/voltage reduction	If there is schedule slack associated with a software task, reduce power by slowing down the clock and reducing the supply voltage.
Guarded evaluation	Latch the inputs of functional units to prevent meaningless calculations when outputs of units are not being used. This increases the power reduction from reduction in strength and operand swapping.

7. Summary

In this chapter, we have dealt with several questions pertinent to the design of software for low power instruction processing systems. Why does software matter in a low power system? How does software contribute to system power? How can one estimate the impact of software design decisions on system power? How can one minimize the software contribution to power dissipation? How can the hardware and software design fit together for a low power system? Several answers to each of these questions have been extracted from recent literature.

The primary objective of this activity has been to help in making software design decisions consistent with the objective of power minimization. Key objectives to be considered are the following. Choose the best algorithm for the problem at hand and make sure it fits well with the computational hardware. Failure to do this can lead to costs far exceeding the benefit of more localized power optimizations. Minimize memory size and expensive memory accesses through algorithm transformations, efficient mapping of data into memory, and optimal use of memory bandwidth, registers and cache. Optimize the performance of the application, making maximum use of available parallelism. Take advantage of hardware support for power management. Finally, select instructions, sequence them, and order operations in a way that minimizes switching in the CPU and datapath.

References

1. Binh, N. N., Imai, M., Shiomi, A., and Hikichi, N. (1995). An instruction set optimization algorithm for pipelined ASIP's. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E78A(12),1707–1714.
2. Catthoor, F., Franssen, F., Wuytack, S., Nachtergaele, L., and DeMan, H. (1994). Global communication and memory optimizing transformations for low power signal processing systems. In *Proceedings,IEEE Workshop on VLSI Signal Processing*, 178–187.
3. Chandrakasan, A., Allmon, R., Stratakos, A., and Brodersen, R. (1994). Design of portable systems. In *IEEE Custom Integrated Circuits Conference*, 259–266.
4. Chandrakasan, A. and others, . (1995). Optimizing power using transformations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(1),12–31.
5. Chou, T. and Roy, K. (1996). Accurate estimation of power dissipation in CMOS sequential circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.
6. Davidson, J. W. and Jinturkar, S. (1994). Memory access coalescing: A technique for eliminating redundant memory accesses. *ACM SIGPLAN Notices*, 29(6).
7. DeGreef, E., Catthoor, F., and DeMan, H. (1995). Memory organization for video algorithms on programmable signal processors. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 552–557.
8. Ellis, S. (1993). The low power Intel486SL microprocessor. In *IEEE COMPCON*, 88–95.
9. Genin, D., DeMoortel, J., Desmet, D., and deVelde, E. V. (1989). System design, optimization and intelligent code generation for standard digital signal processors. In *Proceedings of the International Symposium on Circuits and Systems*, 565–569.
10. Harris, E. P., Depp, S. W., Pence, W. E., Kirkpatrick, S., Sri-Jayantha, M., and Troutman, R. R. (1995). Technology directions for portable computers. *Proceedings of the IEEE*, 83(4),636–657.
11. Hasegawa, A., Kawasaki, I., Yamada, K., Yoshioka, S., Kawasaki, S., and Biswas, P. (1995). SH3: High code density, low power. *IEEE Micro*, 11–19.
12. Hilfinger, P. N., (1993). Silage reference manual, draft release 2.0. http://infopad.eecs.berkeley.edu/~hyper/Hyper/Intro/Silage/reference.ps. University of California, Berkeley.
13. Huang, I.-J. and Despain, A. M. (1994). Synthesis of instruction sets for pipelined microprocessors. In *Proceedings, Design Automation Conference*, 5–11.

14. Lauwereins, R., Engels, M., Peperstraete, J., and Steegmans, E. (1990). GRAPE: A CASE tool for digital signal parallel processing. *IEEE ASSP Magazine*, 7(2),32–43.
15. Lee, M. T.-C. and Tiwari, V. (1995). A memory allocation technique for low-energy embedded DSP software. In *Proceedings of the Symposium on Low Power Electronics*, 44–45.
16. Lee, M. T.-C., Tiwari, V., Malik, S., and Fujita, M. (1995). Power analysis and minimization techniques for embedded DSP software. *Fujitsu Scientific and Technical Journal*, 31(2),215–229.
17. Liao, S., Devadas, S., Keutzer, K., Tjiang, S., and Wang, A. (1995). Code optimization techniques for embedded DSP microprocessors. In *Proceedings of the Design Automation Conference*, 599–604.
18. Lidsky, D. B. and Rabaey, J. M. (1994). Low power design of memory intensive functions. case study: Vector quantization. In *IEEE Workshop on VLSI Signal Processing, Proceedings*, 378–387.
19. Najm, F. N. (1994). A survey of power estimation techniques in VLSI circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4),446–455.
20. Potkonjak, M. and Rabaey, J. M. (1995). Power minimization in DSP application specific systems using algorithm selection. In *Proceedings, International Conference on Acoustics, Speech, and Signal Processing*, 2639–2642.
21. Sato, J., Alomary, A. Y., Honma, Y., Nakata, T., Shiomi, A., Hikichi, N., and Imai, M. (1994). PEAS-I: A hardware/software codesign system for ASIP development. *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Sciences*, E77A(3),483–490.
22. Sato, T., Ootaguro, Y., Nagamatsu, M., and Tago, H. (1995). Evaluation of architecture-level power estimation for CMOS RISC processors. In *Proceedings of the Symposium on Low Power Electronics*, 44–45.
23. Srivastava, M. B., Chandrakasan, A. P., and Brodersen, R. W. (1996). Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1),42–54.
24. Su, C.-L., Tsui, C.-Y., and Despain, A. M. (1994). Low power architecture design and compilation techniques for high-performance processors. In *Proceedings of IEEE COMPCON*, 489–498.
25. Sudarsanam, A. and Malik, S. (1995). Memory bank and register allocation in software synthesis for ASIP's. In *Proceedings of the International Conference on Computer-Aided Design*, 388–392.
26. Tiwari, V. and Lee, M. T.-C. (1995). Power analysis of a 32-bit embedded microcontroller. In *Proceedings of the Asia and South Pacific Design Automation Conference*, 141–148.
27. Tiwari, V., Malik, S., Wolfe, A., and Lee, M. T.-C., (1996). Instruction level power analysis and optimization of software. Accepted for Publication to the Journal of VLSI Signal Processing. http://www.ee.princeton.edu/~sharad/pubs.html.
28. Yao, F., Demers, A., and Shenker, S. (1995). A scheduling model for reduced CPU energy. In *Proceedings of the IEEE 36th Annual Symposium on Foundations of Computer Science*, 374–382.