Efficient Instruction-Level Optimization Methodology for Low-Power Embedded Systems*

Kyu-won Choi and Abhijit Chatterjee
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
{kwchoi,.chat}@ece.gatech.edu

Abstract

In this paper, for low-power embedded systems, we solve the instruction scheduling and reordering problem as a Precedence Constrained Hamiltonian Path Problem for DAGs and the Traveling Salesman Problem (TSP), both of which are NP-Hard [1,2]. We propose an efficient instruction-level optimization algorithm for solving the NP-Hard problem. Minimum spanning tree (MST) and simulated annealing (SA) mechanisms are used for the optimization. We describe the methods for generating the control flow and data dependence graph (CDG), power dissipation table (PDT), and weighted strongly connected graph (SCG) for the instruction-level low-power analysis. In addition, confidence limits with error tolerance are considered for the validation of the optimization. Finally, experimental results that demonstrate the effectiveness and the efficiency of the proposed algorithms are shown.

1 Introduction and Previous Work

Recently, new research directions in reducing power consumptions for embedded systems have begun to address the issues of arranging software at the instruction level to control power dissipation [3,4,5,6,7,8,9,10]. The energy consumed by a processor depends on the previous state of the system and the current inputs. Therefore, it is clear that proper instruction choice and ordering should be considered for low-power embedded systems.

There has been previous research on instruction-level scheduling to reduce total power consumption. In [3], a technique called cold scheduling was combined with Gray code addressing to reduce the switching activity in the control path of high-performance processors. The cold scheduling algorithm works much the same way as traditional list scheduling with the exception that it schedules instructions based on a modified priority scheme.

Another scheduling technique for reducing power consumption is introduced in [4,5]. The goal here is to reschedule code such that instructions are more judiciously chosen as opposed to actually reordering instructions to reduce power. In order to achieve the goal of low-power scheduling, the authors implement a methodology for determining the energy consumption of a single instruction in which they measure the current through the processor when that instruction is executed, using hardware.

In [6], the authors introduce a redundant search space reduction scheme called BARS. Basically, BARS examines all possible schedules of a DAG, and finds the schedule with the lowest cost. The BARS algorithm employs two techniques for runtime efficiency. One technique avoids potentially redundant search, and another technique limits the number of sub-trees to be searched.

In this paper, we propose an efficient instruction-level optimization algorithm for low-power embedded systems. The proposed scheme has the following benefits:

1) The instruction-ordering problem for low power is formulated as a combinatorial graph-search optimization problem, yielding near-optimal results.
2) RTL-level power simulators are used to drive the optimization process.
3) A methodology for validating the results of the optimization process is presented.

2 Key Contributions

In this paper, we tackle the following problems with regard to instruction-level optimization:

1) The choice of instruction sequences is critical to reducing power consumption. The instruction scheduling or reordering problem is equivalent to the Precedence Constrained Hamiltonian Path problem for DAGs and the Traveling Salesman Problem in general, both of which are *NP-Hard*. With n instructions, there are (n-1)!/2 possible instruction sequences. For example, for 11 instructions, there are 16,314,400 possible combinations of the instruction sequences. For large numbers of instructions it is practically impossible to examine all possible instruction sequences to find the optimal sequence for low power. Recent research has demonstrated that design decisions made at the RTL level can have a dramatic impact on the overall power characteristics of the design [3,4.5,7,8]. Therefore, power estimation of assembly instructions at the RTL level is very useful, if accuracy is guaranteed within reasonable simulation time constraints.

* This work is sponsored by DARPA under the grant # E21-F48.

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.

2) Due to the large search space, the optimality of the search algorithm should be validated at the very least in a probabilistic sense.

3 Problem Description

The general procedure for instruction-level analysis is shown in Figure 1(a-d):

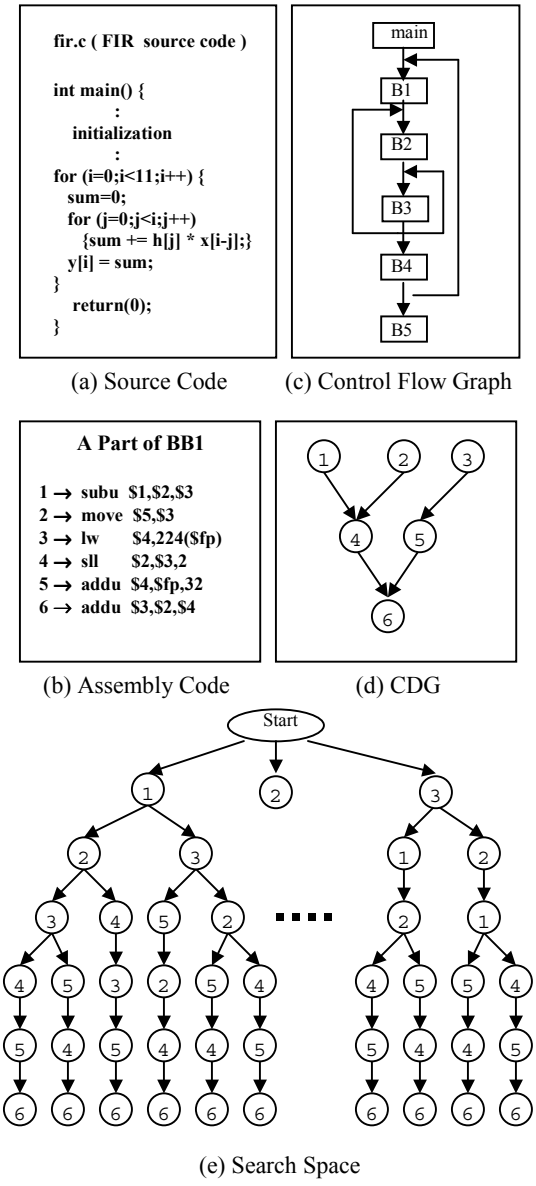
1) Assembly instructions are assembled from the source program.
2) The assembly code is decomposed into Basic Blocks (BB).
3) Control and data dependency graph (CDG) is constructed.

After obtaining the CDG, instruction scheduling is performed for each BB and optimal instruction code for low power is provided.

The main problem in instruction-level scheduling is that the search space of the possible instruction sequences is very large. The search space for six instructions of an FIR filter source code for one BB is shown in Figure 1(e). Even though only six instructions are considered, the search space is large.

Figure 2(a) shows a power dissipation table (PDT) which shows the average power consumed when an instruction in the left most column is followed by an instruction in the topmost rows. The control flow and data dependency graph (CDG) of Figure 2(b) shows the inter-dependency of instructions. For example, instruction ④ can be executed only after instruction ① and ② have both been executed, while instructions ①,②, and ③ may be executed in any order. The weighted strongly connected graph (SCG) of Figure 2(c) contains all the edges of the CDG of Figure 2(b) and some additional edges. If the order in which any two instructions are executed can be interchanged without violating the constraints of the CDG, then an additional edge is inserted between the graph nodes (vertices) corresponding to the two instructions. For example, the instruction ④ and ⑤ of the CDG in Figure 2(b) can be interchanged. Hence, in Figure 2(c), there is an edge between the nodes ④ and ⑤. Each edge of the SCG is assigned a weight corresponding to the power consumed by running the pair of instructions repeatedly. It is assumed that the order in which the instruction pair is executed does not matter.

Once the SCG is obtained, a minimum spanning tree algorithm in Figure 2(d) is used to obtain an initial sequence for the optimization. After the MST is performed, local improvement heuristics are conducted by using simulated annealing to obtain an optimal (least cost) Hamiltonian Tour of all the vertices of the SCG as shown in Figure 2(e). This tour gives the optimal sequence of instructions from the viewpoint of power. In this case, the optimal sequence is ③,②,①,④,⑤,⑥.



(a) Source Code (c) Control Flow Graph

(b) Assembly Code (d) CDG

(e) Search Space

Figure 1. CDG and Search Space Example

4 Optimization Procedure

The instruction-level optimization methodology is given below:

Step1: Assemble the source code
Step2: Decompose into basic blocks
Step3: Construct CDG for each basic block
Step4: Generate PDT from power simulation
Step5: Build weighted SCG from CDG and PDT
Step6: Solve TSP problem to find optimal or near optimal solution
Step7: Validate the optimization algorithm
Step8: Generate instruction sequence for low-power

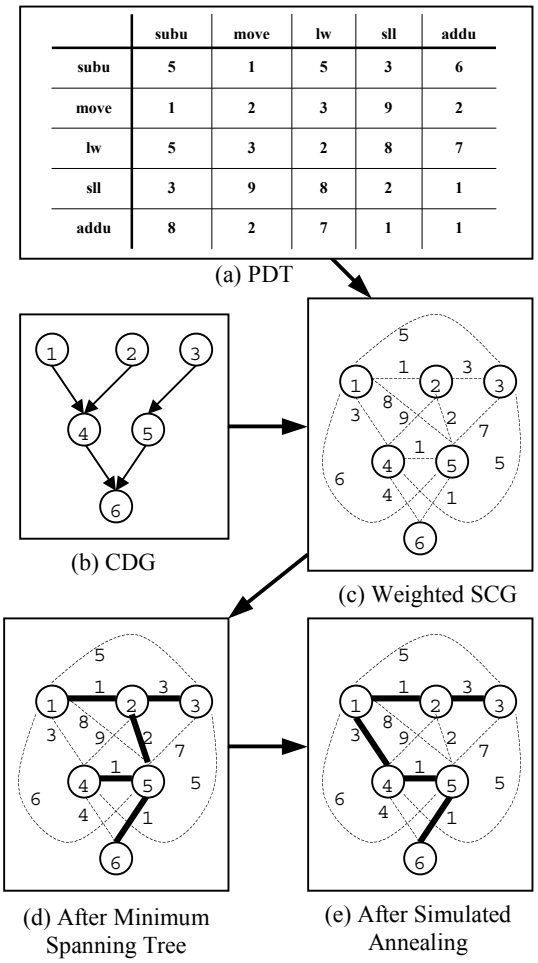


Figure 2. Our Approach

In the following, we discuss specific steps of the optimization procedure.

4.1 CDG Construction

First, we review basic graph notations that are used in this paper. A graph ***G*** consists of a set of nodes ***V***, and a set of edges ***E***, ***G*** = <***V,E***>. The degree of a node is the number of neighbors adjacent to it. We write ***X*→*Y*** when <***X,Y***> ∈ ***E*** in a directed graph. We define ***PRED(X)*** as the set of the all predecessors of node ***X***, and ***SUCC(X)*** as the set of all successors of ***X***. If ***X*→*Y***, then ***X* ∈ *PRED(Y)*** and ***Y* ∈ *SUCC(X)***. The ***indegree*** of a node ***X*** is the cardinality of ***PRED(X)*** and the ***outdegree*** of a node ***X*** is the cardinality of ***SUCC(X)***. The ***indegree(X)*** is the number of edges of the form ***W*→*X***, and the ***outdegree(X)*** is the number of edges ***X* → *Y***. If ***W*→*X*** , then ***W* ∈ *INDEG(X)*** and ***X* ∈ *OUTDEG(W)***. So, the ***OUTDEG(W)*** is a sub set of ***SUCC(W)***, and the ***INDEG(X)*** is the sub set of ***PRED(X)***. The notation ***X* *→*Y*** is to mean that there is a path from ***X*** to ***Y***. The control flow and data dependency graph specifies the execution order among the components for given instructions to obey the original program semantics. Figure 3 shows the CDG construction algorithm.

Algorithm 1 Algorithm for CDG Construction

Input: Instruction-level source code
Output: Control and Data flow DAG representation
Definition: **idn:x** : id number (**idn**) for each instruction **x**
src_pre(x) : source operands of previous instruction of **x**
des_pre(x) : destination operands of previous instruction of **x**
src(x) : current source operands of **x**
des(x) : current destination operand of **x**

step 0: Set identity number (**idn:x**) for each instruction **x**.
step 1: Calculate total BB number and store the start **idn** of each **BB**.
step 2: For each **x** in **BB**, visit **x**, identify **src_pre(x)**, **des_pre(x)**, **src(x)**, and **des(x)**.
step 3: IF there is any **y** in **des_pre(x)** associated with **src(x)**,
THEN create an <**x,y**>∈**E** in **G=<V,E>** and remove **y** from **des_pre(x)**
step 4: IF there is any **y** in **des_pre(x)** associated with **des(x)**,
THEN create an <**x,y**>∈**E** in **G=<V,E>** and remove **y** from **des_pre(x)**
step 5: IF there is any **y** in **src_pre(x)** associated with **des(x)**,
THEN create an <**x,y**>∈**E** in **G=<V,E>** and remove **y** from **src_pre(x)**
step 6: IF there is any **x** in **BB** to be scheduled,
THEN go to **step 2**.
ELSE generate CDG graph and return.

Figure 3. CDG Construction Algorithm

4.2 PDT Generation

The power dissipation table describes the power consumption due to the execution of an instruction itself and a pair of instructions. We use a cycle-accurate data path energy model at the RTL level. A cycle-by-cycle instruction-level simulator is used to collect the switching activity at all latches in the data path during execution of the program. The SimpleScalar tool set [11], version 3.0 with RTL level power model is used. Our target machine has a five-stage pipelined data path and the functional units are 32-32bit general registers, I-cache, D-cache, divider, multiplier, ALU, shifter, and control units.

Details of generating infinite loops to measure the power dissipation due to an instruction itself and an instruction pair are shown in Figure 4. To avoid any corruption due to loop overhead, we repeat the instruction pair 200,000 times with a *nop* operation.

```
main:
:
lw    $2,228($fp)
move $5,$4
nop
lw    $2,228($fp)
move $5,$4
nop
lw    $2,228($fp)
move $5,$4
nop
… 200,000 times
:
```

Figure 4. PDT Generation Example

4.3 SCG Generation

We define a strongly connected graph that is a set of nodes S such that there is $S_1 *\rightarrow S_2$ for any two nodes $S_1, S_2 \in S$. As shown in Figure 2(c) in Section 3, the weighted SCG is a sub set of all possible instructions corresponding to the CDG. The SCG Generation algorithm is shown in Figure 5.

Algorithm 2 Algorithm for SCG Generation

Input: **CDG** stream
Output: **SCG** DAG representation (summation of **SCC**s)
Definition: **indegree(X)** : the number of edges of **W→X**
outdegree(X) : the number of edges of **X→Y**
INDEG(X) : **If W→X , then W ∈ INDEG(X)**
OUTDEG(X) : **If X→Y , then Y ∈ OUTDEG(X)**
SCC(X) : strongly connected components of **X**
means that the components in **SCC(X)**
may be followed by an instruction **X**
(i.e., **SCC(X)= [Y, Z]:Y,Z** can be followed by **X**)

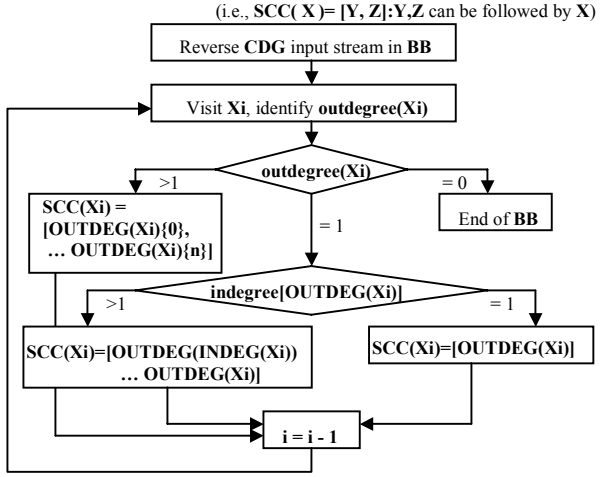


Figure 5. SCG Generation Algorithm

4.4 TSP Realization

In the general form of the *traveling salesman problem*, we are given a finite set of points V and a cost c_{uv} of travel between each pair $u.v \in V$. A tour is a circuit that passes exactly once through each point in V. The TSP is to find a tour of minimal cost. The TSP can be modeled as a graph problem by considering a complete graph $G = <V,E>$, and assigning each edge $u.v \in E$ the cost c_{uv}. A tour is then a circuit in G that meets every node. The tours are sometimes called *Hamiltonian Circuits*. The TSP is one of the best-known problems of combinatorial optimization. A nice collection of papers tracing the history and research on the problem can be found in [12]. No polynomial-time algorithm is known for solving the TSP in general. Indeed, it belongs to the class of *NP-Hard* problems.

Heuristics are methods that cannot be guaranteed to produce optimal solutions, but which produce fairly good solutions at least some of the time. For the TSP, there are two different types of heuristics. The first attempts to construct a good tour from scratch. The second tries to improve an existing tour, by means of local improvements. In practice it seems very difficult to get a really good tour construction method [12]. In this paper, as a construction method, we use a minimum-cost spanning tree by using Prim's Algorithm [13], and as a improvement method, we use ***2-optimal*** and ***3-optimal*** local search heuristics by using simulated annealing. Prim's algorithm is used for computing a minimum spanning tree. It builds upon a single partial minimum spanning tree, at each step adding an edge connecting the vertex nearest to but not already in the current partial minimum spanning tree. If Prim's Algorithm is applied to the graph of Figure 6(a) with starting vertex ***a***, edges are chosen in the order ***ab***, ***af***, ***ac***, ***cd***, ***dg***, ***de*** as shown in Figure 6(b).

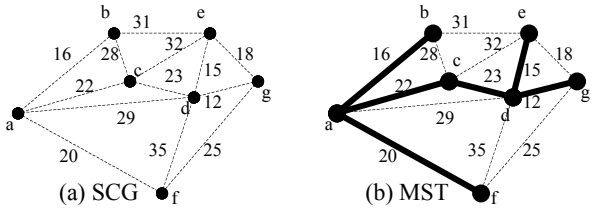


Figure 6. Prim's Algorithm Example for MST

The MST solution is used as a starting point for further optimization using heuristics. In this paper, we use ***2-optimal*** and ***3-optimal*** mechanisms for the improvement heuristic, using simulated annealing with the reasonable cooling parameters as the optimization engine. The ***2-optimal*** heuristic proceeds by considering each non-adjacent pair of edges of tree T in turn. If these edges are deleted, the T breaks up into two paths T_1 and T_2. There is a unique way that these two paths can be recombined to form a new tour T'. If $c(T') < c(T)$, then we replace T with T' and repeat the procedure. As shown in Figure 7, if $c(T') > c(T)$ for every choice of pairs of nonadjacent edges, then T is ***2-optimal*** and we terminate the procedure.

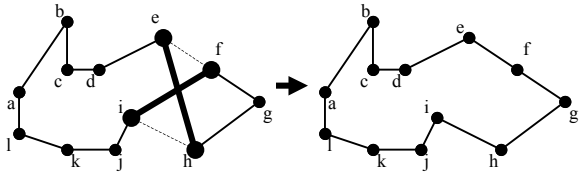


Figure 7. 2-Interchange Local Search Heuristics

The ***2-optimal*** algorithm can be generalized naturally to a ***k-optimal*** algorithm, wherein we consider all subsets of the edge-set of a tour of size of k, or size at most k, remove each subset in turn, then see if the resulting paths can be recombined to form a tour of lesser cost. The problem is that the number of subsets grows exponentially with k, and we soon reach a point of diminishing return. For this reason, ***k-optimal*** for $k>3$ is rarely used. Figure 8 shows ***3-optimal*** example. In this paper we use both the ***2-optimal*** and the ***3-optimal*** heuristics for optimization.

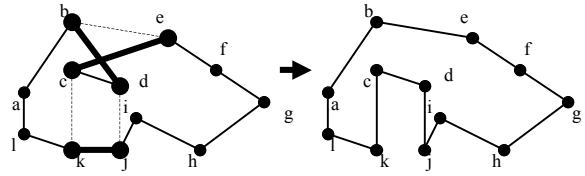


Figure 8. 3-Interchange Local Search Heuristics

5 Validation of Optimization Methodology

Let us designate the sample space $n=(k-1)!/2$ for k instructions in a **BB**. Let us define v as an input vector for an instruction sequence, define random variable $W(v)$ to be the power dissipation of the system when v is applied. If we choose j input vectors $v_1, v_2,\ldots,v_j$ randomly and exclusively, the power dissipation obtained in each trial is also an independent random variable (RV) $W_j=W(v_j)$, $j=1,\ldots,n$. Since the W_j are independent, the set $\{W_1, W_2,\ldots,W_j\}$ constitutes a random sample. We define a new RV $X_j=min(W_1, W_2,\ldots,W_j)$, so that X_j is the minimum power dissipation over j trials.

Let us designate the jth ordered observation ${}_oX_j$ and the fraction of the population below this observation as $P_j=F({}_oX_j)$. Since the value of ${}_oX_j$ varies from sample to sample, P_j is a random variable. Thus we must consider the distribution of P_j. We will now proceed with the development of the distribution for P_j.

Given an ordered random sample ${}_oX_1, {}_oX_2, \ldots, {}_oX_n$ of size n from a population having a cumulative distribution function $F(X)$, where X is continuous, we try to find estimators for $F({}_oX_1), F({}_oX_1), \ldots, F({}_oX_n)$. Let us define ${}_nP_j=F({}_oX_j)$, where ${}_nP_j$ is the probability of failing to find an optimal minimum sequence X_n prior to the jth ordered observation in a sample space n. Then, we have the $F({}_oX_j)$ probability for $j-1$ outcomes, $f({}_oX_j)dX$ for 1 outcome, and $1- F({}_oX_j)$ for $n-j$ outcomes. Clearly, the multinomial distribution is applicable in this situation. Recall that the multinomial distribution is given by

$$P(y_1,y_2,\ldots,y_k)=\frac{m!}{y_1!y_2!\ldots y_k!}(\Theta_1)^{y_1}(\Theta_2)^{y_2}\ldots(\Theta_k)^{y_k} \qquad (1)$$

Where, the Θ_i is the probability of obtaining the ith outcome and y_i is the RV representing the number of outcomes of the ith type.

Therefore, the probability of $j-1$ outcomes with $F({}_oX_j)$, 1 outcome with $f({}_oX_j)dX$, and $n-j$ outcomes with $1- F({}_oX_j)$ is given by

$$\frac{n!}{(j-1)!1!(n-j)!}[F({}_oX_j)]^{j-1}f({}_oX_j)dX[1-F({}_oX_j)]^{n-j} \qquad (2)$$

This is precisely the distribution of ${}_oX_j$, the value of the jth odered observation.

Now we know that

$$dF({}_oX_j)=f({}_oX_j)dX=d_np_j \qquad (3)$$

Hence the probability element becomes

$$g({}_np_j)d_np_j=\frac{n!}{(j-1)!(n-j)!_n}{}_np_j^{j-1}(1-{}_np_j)^{n-j}d_np_j(0\le{}_np_j\le 1) \qquad (4)$$

This distribution is commonly termed the rank distribution. Actually the p.d.f. of the RV ${}_nP_j$ is the well-known beta distribution [14]. Figure 9 shows the p.d.f. for different values of j for a sample of size n=10.

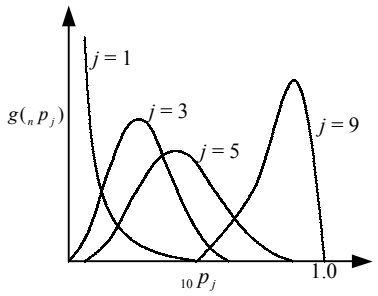


Figure 9. The rank distribution for a sample of size 10

Recall that the ${}_nP_j$ is the probability of failing to find an optimal minimum sequence X_n prior to the jth ordered observation in a sample space n. Let us define that ε is the error tolerance with a range of $0\le\varepsilon\le1$ and α is confidence limit with range $0\le\alpha\le1$.

$$P\{{}_np_j\le\varepsilon\}\ge\alpha \qquad (5)$$

This equation describes the method used to estimate the confidence limit of the optimization with error tolerance ε. For example, in order to get 95% confidence with less than 5% error tolerance for any optimization value, it is estimated as follows:

1) Estimate j trial number from the $P\{{}_np_j\le0.05\}\ge0.95$ in the sample space n
2) Generate j input instruction sequences randomly and exclusively
3) Find the minimum value among all j trial sequences
4) Compare the minimum value (minimum power) and TSP optimized value
5) If two values are close, we can say that the TSP optimization has 95% confidence limit with 5% error tolerance.

6 Experiments and Results

All simulations are done by using the SimpleScalar tool set [11], version 3.0 with RTL level power model. The target machine includes a five-stage pipelined data path, 32-32bit general registers, I-cache, D-cache, divider, multiplier, ALU, shifter, and control units. The simulated annealing parameters are tuned for 95% confidence limit and 5% error in Equation (5). We assume a fixed clock frequency, fixed supply voltage, and 0.35 micron technology. So, the terms "power" and "energy" can be used interchangeably because the clock frequency is fixed. Moreover, the power is almost proportional to the switching capacitance, due to the fact that over 90% of the total power dissipation is the switching power in well-designed circuits [15]. We analyze the power consumption without any schedule and with the optimal schedule when running several embedded applications. Figure 10 shows an optimal instruction sequence example for an FIR filter application. When one basic block is optimized, 8.5 % of the total power can be reduced with our scheme. Figure 11 presents power savings in each component of the target system. Table 1 shows that a maximum 29% of the total power can be saved with our algorithm for about 95% confidence limit with 5 % error tolerance.

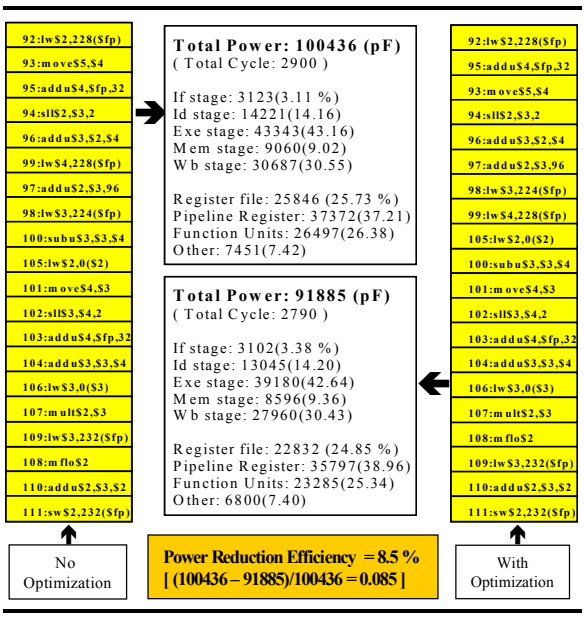
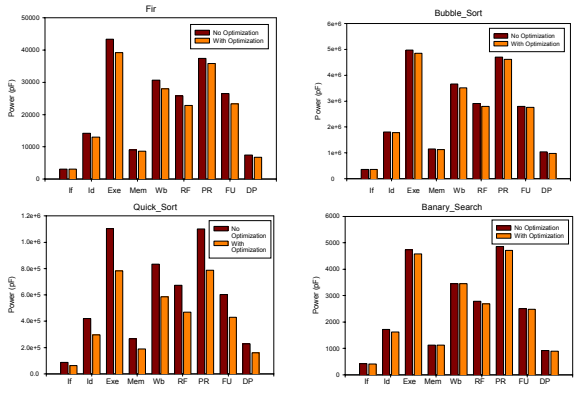


Figure 10. Scheduling Example for one BB of FIR Source



Note: If: Instruction Fetch, Id: Instruction Decode, Exe: Execution, Mem: Memory access, Wb: Wright back, RF: Register file, PR: Pipeline Register, FU: Functional Units, DP: Other Data Paths

Figure 11. Power Savings for Each Component

Table 1. Total Power Saving Results for Some Applications

	Total Power / (Cycles) without Scheduling (pF)	Total Power / (Cycles) with Scheduling (pF)	Confidence Limit / Error Rate (%)	Power Reduction (%)
FIR Filter	100436.35 / (2900)	91885.50 / (2790)	≈ 95 / ≈ 5	**8.52**
Bubble Sort (100 Samples)	11948917.22 / (368100)	11627652.60 / (360675)	≈ 95 / ≈ 5	**2.68**
Quick Sort (100 Samples)	2711445.75 / (85502)	1919845.60 / (61170)	≈ 95 / ≈ 5	**29.19**
Heap Sort (100 Samples)	3615121.35 / (101136)	3373014.48 / (96329)	≈ 95 / ≈ 5	**6.69**

7 Conclusion and Future Work

In this paper, we show that the instruction-level optimization problem can be solved using combinatorial methods. An efficient instruction-level optimization methodology and a validation method for the optimization are proposed. Experimental results show the effectiveness of the new approach.

The goal of this paper is to present a methodology for developing and validating an instruction-level low-power optimization framework for a given instruction set architecture. Future work will include global compiler-directed solutions for low-power embedded systems that will also consider the power benefits of voltage and frequency scaling.

References

[1] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver, *Combinatorial Optimization*, Wiley-Interscience Series in Discrete Mathematics and Optimization, pp241-271, 1998.

[2] V. Jain, S. Rele, S. Pande, and J. Ramanujam "Code restructuring for improving execution efficiency, code size and power consumption for embedded DSPs", *12th International Workshop on Languages and Compilers for Parallel Computing*, 1999.

[3] C.-L. Su, C.-Y. Tsui, and A. M. Despain, "Low power architecture design and compilation techniques for high-performance processors," *Proc. IEEE CompCon'94*, pp. 489-498. Feb., 1994.

[4] V. Tiwari, S. Malik, and A. Wolfe, "Instruction Level Power Analysis and Optimization of Software," *Journal of VLSI Signal Processing*, pp.1-18, 1996.

[5] V. Tiwari, S. Malik, and A. Wolfe, "Compilation Techniques for Low Energy: An Overview," *IEEE Symposium on Low Power Electronics*, October 1994.

[6] H. Tomiyama, T. Ishihara, A. inoue, and H. Yasuura, "Instruction Scheduling to Ruduce Switching Activity of Off-Chip Busses for Low-Power Systems with Caches," *Proc. IEICE*, pp. 2621-2629. Dec., 1998.

[7] C. Lee, J.K. Lee, and T.T. Hwang, "Compiler optimization on Instruction Scheduling for Low Power," *ISSS2000*, pp.20-22, Oct., 2000.

[8] K.Roy and M.C. Johnson, "Software design for low power," *Technical report at Purdue Univ.*, 1996.

[9] C.-L. Su, C.-Y. Tsui, and A. M. Despain, "Low power architecture design and compilation techniques for high-performance processors," *Proc. IEEE CompCon'94*, pp. 489-498. Feb., 1994.

[10] G. Lakshminarayana, A. Raghunathan, N.K. Jha, "Incorporating Speculative Execution into Scheduling of Control-Flow-Intensive Designs," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol., 19, no. 3, pp. 308-324Mar., 2000.

[11] D. Burger and T.M. Austin, "SimpleScalar Tool Set," *Univ. of Wisconsin-Madison Computer Science Dept., Tech. Teport #1342*, Jun., 1997.

[12] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D. Shmoys, *The Traveling Salesman Problem*, Wiley, Chichester, 1985.

[13] R.C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal36*, pp1389-1401, 1957.

[14] K.C. Kapur and L.R. Lamberson, *Reliability in Engineering Design*, John Wiley & Sons, 1977.

[15] T.D. Burd and R.W. Brodersen, "Energy efficient CMOS microprocessor design," *Proceedings 28th HICSS Conference*, vol. I, pp. 288-297, Jan., 1995.