Global Register Allocation for Minimizing Energy Consumption

Yumin Zhang Xiaobo (Sharon) Hu Danny Z. Chen
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
{yzhang1, shu, chen}@cse.nd.edu

Abstract

Data referencing during program execution can be a significant source of energy consumption especially for data-intensive programs. In this paper, we propose an approach to minimize such energy consumption by allocating data to proper registers and memory. Through careful analysis of boundary conditions between consecutive blocks, our approach efficiently handles various control structures including branches, merges and loops, and achieves superior allocation results for the whole program. The computational cost of our approach for solving the global register allocation problem is rather low comparing with known approaches while the quality of our results is very encouraging.

1 Introduction

Accessing registers rather than memory generally takes less time and consumes less energy. Due to the limited number of registers in a processor, register allocation plays an important role for achieving high performance and low energy consumption in program execution. Most existing register allocation techniques have focused on reducing program execution time [2, 4]. Since energy consumption depends on data correlation, an optimal-time register allocation does not necessarily lead to the least amount of energy consumption. Approaches dealing with the optimal-energy register allocation problem have been proposed in [3, 5]. But all these algorithms only consider variables within a piece of straight-line code called a basic block. Complex control structures in a program is not considered by these approaches.

In this paper, we extend the work in [5, 8] by investigating the global register allocation for a program containing branches, merges and loops. We have developed algorithms for handling the boundary conditions between consecutive blocks. Our approach can produce superior allocation results efficiently. High compiling speed can improve software productivity and is important for dynamic compiling applications [7].

2 Preliminary

The total energy consumed by a program is the sum of energy consumed by register accesses and memory accesses. Different energy models, such as the static and activity-based energy models [3, 5], affect the computation of the energy consumption of program execution. Under the static energy model, optimizing energy consumption is equivalent to minimizing the number of accesses to memory, which is also equivalent to optimizing execution time. For the activity-based energy model, the energy consumed by a program is dependent on the switched capacitance of the storage elements. The sum of Hamming distance between two subsequent data items which share the same storage location has been used to compute the total switched capacitance of storage elements. To simplify the problem formulation, we will employ the activity-based energy model for assessing the register file and static energy model for accessing memory. (Adopting the activity-based model for memory is a simple extension to the algorithm discussed later in this paper.)

Let V_R (resp., V_M) be the set of variables assigned to registers (resp., memory), and $e^R_{r/w}$ (resp., $e^M_{r/w}$) be the energy consumption of a read/write access to register file (resp., memory). Let $\overline{H}(v_i, v_j)$ be the average Hamming distance between v_i and v_j, C^R_{rw} be the average switched capacitance per bit by the register access, and V be the operational voltage. Then, the total energy consumption of a program can be calculated as $E = N^M_r e^M_r + N^M_w e^M_w + \sum_{v_i \to v_j, v_i, v_j \in V_R} \overline{H}(v_i, v_j) C^R_{rw} V^2$, where N^M_r and N^M_w are the total numbers of read and write accesses to memory, and $v_i \to v_j$ indicates that a register content switches from v_i to v_j.

Techniques for minimizing energy consumption for a single basic block in a program based on the above model were proposed by Gebotys in [5]. By cleverly modeling the lifetime ranges of the variables in a basic block, the author formulates the problem as a minimum-cost network flow problem. Hence, an optimal solution can be obtained in polynomial time. However, programs from real applications unavoidably contain branches and some other control structures. Without considering the relations between different blocks, simply applying the approach in [5] to each basic block may lead to many unnecessary memory/register references at blocks boundaries [9]. Enumerating all possible execution paths in a given program and applying the algorithms in [5] to each path is not only computationally expensive but also involves resolving conflicting assignments. In the rest of the paper, we present an efficient heuristic which can find a nearly optimal solution for low energy register allocation for programs containing branches, merges and loops.

3 Our Approach

In the following, we outline our heuristic algorithm which uses an iterative approach to handle programs containing complex control structures. Our approach processes blocks one by one in a predefined order (to be discussed later). The allocation within each block is modeled as a network flow problem similar to [5]. However, our network flow graph and its associated parameters such as arc costs are defined such that they reflect the allocation results of other blocks. The main advantage of our approach is that it allows information from different blocks to propagate in order to get superior allocation results for the whole program. The single direction of the allocation result propagation avoids the excessive computational cost and hence makes the approach very efficient.

The order for the block-by-block allocation is the topological order of blocks in the control-flow-graph (CFG) of a program. This order guarantees that when a block is processed, all the parent blocks (immediate ancestor nodes in the CFG) have been processed. (Loop is treated specially). The cost on the arcs of the network flow graph for a block is then defined by the boundary condition between the block under consideration and the allocation results from its parent blocks. The CFG corresponding to a program with loops may contain back edges. But those back edges can be identified and eliminated to obtain the topological order.

The main challenge is how to define the cost on arcs in the network flow problem to capture the allocation information from parent blocks. The problem formulation for programs with the branch type control structure is given in [8]. In this paper, we focus on how we formulate the network flow problem for the merge and loop control structures to get superior allocation results.

Before we discuss the merge and loop cases, let us first briefly review the construction of the network flow graph for a basic block [5]. Each variable is represented by its lifetime interval which begins at the time point $t_s(v)$ when the variable is defined and ends at the time point $t_f(v)$ when the variable is last used. We define the critical set of variables as the set that the number of variables with overlapping lifetime in the set is equal to or greater than the total number of available registers, k. A complete bipartite graph is formed between the variables in consecutive critical sets and the variables that begin or end in the middle of two consecutive critical sets. A complete bipartite graph is also formed between the source node, S, and the first critical set. The sink node, T, and the last critical set is connected in the same way.

3.1 Register Allocation for the Merge Structure

The fact that the allocation results from different parent blocks may be different makes the merge case more difficult to handle than the branch case. We first modify the network flow graph as follows. Between the source node and the nodes in or before the first critical set we introduce k more nodes, $n_r(j), j = 1, 2, \dots, k$, each of which corresponds to a register. New arcs are introduced correspondingly. Figure 1 shows the new network flow graph for an example block B in the merge structure. Assume there are two machine registers available in the example. The $n_s(v)$ and $n_f(v)$ in Figure 1 correspond to the begin time, $t_s(v)$, and end time, $t_f(v)$, of variable v. The cost on the newly introduced arcs which go out from the newly introduced register nodes are defined carefully to capture not only the execution probability of each parent blocks but also the allocation information of all the parent blocks, no matter how many of them, that merge into the block being considered. The above cost definition also makes our network flow problem formulation for the merge structure consistent with the formulation for the branch structure [8]. For detailed cost definition on these newly introduced arcs going out from the register nodes, see [9].

To minimize the energy consumption for a block with more than one parent blocks merging into it, all execution scenarios for this block should be considered. Thus, the objective for the flow problem should take into account the probabilities of executing different parent blocks. For block B, denote the probability of executing B as $P(B)$. Then, the objective can be formulated as

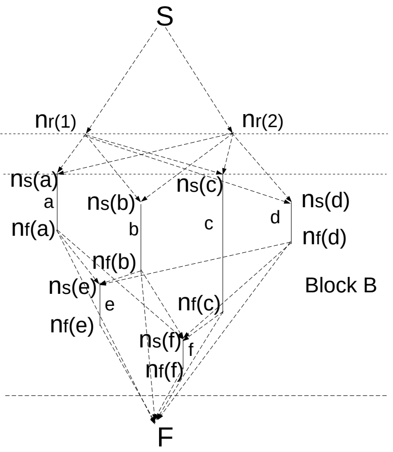

Figure 1: The network flow graph for the merge structure

$$\begin{aligned}\text{minimize:} \quad E = {} & P(B)\sum_{v\in B}(e_r^M + e_w^M) \quad - \\ & \sum_{v\in B|t_s(v)<t_s(B)} \Big(P(v,M,\sum_{j=1}^{m} B^{p_j})e_w^M - P(v,R,\sum_{j=1}^{m} B^{p_j})\lambda e_{rv}^R\Big) - \\ & P(B)\sum_{v\in B|t_f(v)>t_f(B)} e_r^M - P(B)\sum_{a(p,q)\in A} c(p,q)\cdot x(p,q) \qquad (1)\end{aligned}$$

where

$$\lambda = \begin{cases} 1 & \text{if the static energy model is used} \\ 0 & \text{otherwise} \end{cases}$$

A is the set of all arcs, $a(p,q)$, and $P(v,R,\sum_{j=1}^{m} B^{p_j})$ (resp., $P(v,M,\sum_{j=1}^{m} B^{p_j})$) is the total probability of v allocated to a register (resp., memory) in all the parent blocks.

The first three terms in the above objective function (1) are the amount of energy consumed by block B if all the variables live in B are assigned to memory, while the last term represents the energy saved by allocating certain variables to registers. For the activity-based energy model (i.e., $\lambda = 0$), the energy consumption due to register references is computed by the switched capacitance between two consecutive accesses and is captured by the cost, $c(p,q)$, associated with corresponding arcs in the last term of (1). The values of $x(p,q)$ are unknown and to be determined. If $x(p,q) = 1$ and arc $a(p,q)$ corresponds to a variable v, then v will be assigned to a register. As we point out earlier, the values of $c(p,q)$ are dependent on the types of arcs associated with them. All different arcs are categorized and corresponding cost is defined in [9].

Applying a network flow algorithm [6] to our network flow problem instance, the value of each $x(p,q)$ can be obtained in $O(kn^2)$ time for the block B, where k is the number of available registers and n is the number of variables in B. If the resulted x value of an arc associated with a variable is one, the variable is assigned to the appropriate register based on the flow information. The above formulation can then be applied to each basic block in the program control flow graph in the topological order.

3.2 Register Allocation for the Loop Structure

In this section, we extend our approach to handle programs with loops. The unique challenge here is that a loop block itself is also one of its parent blocks. One way to solve the allocation problem for loops is to modify the flow formulation presented above. However, the resulting problem becomes an integer quadratic programming problem, which is expensive to solve. We propose an efficient heuristic approach, which generally produces superior allocation results for loops.

The basis for our approach is the following observations. In general, a loop body is executed frequently. The assignments for

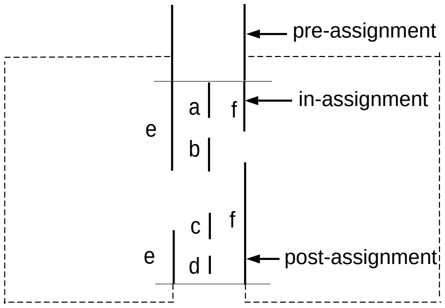

Figure 2: A loop with one basic block

a loop variable at different positions in a loop are referred as pre-assignment, in-assignment, and post-assignment, respectively, as shown in Figure 2. If a loop variable's in-assignment is different from its post-assignment, as many switchings as the number of loop iterations would be needed. Hence, reducing the number of loop variables with different in-assignment and post-assignment would tend to result in better allocations for the program.

Our algorithm employs an iterative multi-phase approach. With the first allocation attempt, ϕ_0, we compute the allocation for the loop block by simply using the results from all the other parent blocks except the loop itself. If there is no loop variable whose in-assignment is different from its post-assignment, the optimal allocation is found and the algorithm exits. Otherwise, we perform two more allocation phases: the second phase (ϕ_1) and the third phase (ϕ_2). In ϕ_1, we let the *in-assignment* of each loop variable to be the value obtained from ϕ_0 and solve the resultant allocation problem after modifying the cost on those arcs to the sink node accordingly. In ϕ_2, we fix the *post-assignment* of each loop variable to the value obtained in ϕ_0 and do the allocation after the cost on arcs from the source node is modified accordingly. The better one from ϕ_1 and ϕ_2 is kept as the best allocation result found so far. Further improvement could be obtained if more iterations were carried out by treating the current best allocation as the one obtained from ϕ_0 and following the same procedure as used in ϕ_1 and ϕ_2. We have proved that the optimal allocation for a loop block is obtained if our algorithm exits after the initialization phase ϕ_0 [9]. Furthermore, we have shown that if the second and third phase are needed, their results always improve that from ϕ_0 [9].

Our approach is essentially a hill-climbing technique and could settle to a suboptimal solution. However, observe that our incremental improvement is based on the optimal allocation for the loop body itself and that there are usually only a small subset of the variables in a loop are loop variables. Therefore, our approach tends to have a good chance to obtain the optimal allocation. Our experimental results also support this observation.

Our algorithm can be readily extended to handle programs with non-single block loops. If a loop contains branches and merges, each phase in our algorithm needs to solve the allocation problem for each block in the loop following the process discussed previously. In ϕ_1 (resp., ϕ_2), the in-assignments (resp., post-assignments) of loop variables obtained from ϕ_0 are used to compute the necessary cost or probability values. For nested loops, the inner-most loop is allocated first and the result is used by the outer loops.

More details about our approach are available upon request [9].

4 Experimental Results and Discussion

To compare our approach with existing approaches, we implemented a graph-coloring allocator and our block-by-block allocator in the Tiger compiler [1]. Since graph-coloring based techniques cannot readily incorporate the activity-based energy model, the comparison is done only based on the static energy model. Note that our results cannot be directly compared with that in [5] since the latter cannot handle programs with complex control structures.

The allocation results by the two allocators are compared for several real programs. These programs contain complex control structures and Table 1 summarizes their relevant parameters. The numbers of memory references by the two allocators and the improvement of our approach are summarized in Table 2.

Table 1: Number of memory references by two approaches

Example Programs	# of Registers available	# of variables to be allocated	# of data references
sort	16	106	1218
factorial	14	36	309
branch	14	28	67

Table 2: Improvement of our algorithm over graph-coloring

Programs	Our approach # of memory reference	Graph-coloring # of memory references	Improvement # of memory references (%)
sort	0	284	23.3
factorial	0	92	29.8
branch	0	16	23.9

Considering that one memory reference usually consumes more than 10 times of energy than a register reference, our algorithm can achieve more than 20% improvement in the energy consumption. Furthermore, our approach is simple to use and the running time is polynomial. More experiments with larger code sizes are being conducted. We are also investigating techniques for interprocedure register allocation to reduce energy consumption.

5 Acknowledgment

This research was supported in part by the National Science Foundation under Grant CCR-9623585 and MIP-9701416, and an External Research Program Grant from Hewlett-Packard Laboratories, Bristol, England.

6 References

[1] A.W. Appel, *Modern Compiler Implementation in C*, Cambridge University Press, 1997.

[2] G.J. Chaitin,"Register allocation and spilling via graph coloring," *ACM SIGPLAN Symposium on Compiler Constructions*, 1982, pp. 98-101.

[3] J. Chang and M. Pedram,"Register allocation and binding for lower power," *DAC'95*, pp. 29-35.

[4] F.C. Chow and J.L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4, October 1990, pp. 501-536.

[5] C.H. Gebotys, "Low energy memory and register allocation using network flow," *DAC'97*, pp. 435-440.

[6] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, 1976.

[7] O. Traub, G. Holloway, and M. Smith, "Quality and speed in linear-scan register allocation," *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, 1998, pp. 142-151

[8] Y. Zhang, X.S. Hu, and D.Z. Chen, "Low energy register allocation beyond basic blocks," *Proceedings of IEEE International Symposium on Circuits and Systems*, 1999.

[9] Y. Zhang, X. Hu, and D. Chen,"Global Register Allocation for Minimizing Energy Consumption,"Technical Report TR99-5, Department of Computer Science and Engineering, University of Notre Dame, 1999.