Low Energy Register Allocation Beyond Basic Blocks

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
An approach of doing register allocation beyond basic blocks for low energy is presented in this paper. With careful analysis of boundary conditions between consecutive blocks, our approach achieves the allocation results benefiting the whole program. By doing the allocation block by block, we avoid excessive computational cost.

1 Introduction

Today’s high demand of portable electronic products makes low energy consumption as important as high speed and small area in computer system design. Even for non-portable high performance systems, lower power consumption design helps to decrease packing and cooling cost and increase the reliability of the systems [24]. A lot of research has been done in improving the power consumptions of various components in computer systems [8, 12, 18, 19, 24]. In particular, it is well recognized that access to different levels of storage components, such as register, cache, main memory and magnetic disk, differs dramatically in speed and power consumption [8, 16]. Hence allocation of variables in a program to different storage elements plays an important role toward achieving high performance and low energy consumption [12]. With today’s IC technology and computer architecture, memory read/write operations take much more time and consume much more power than register read/write operations [3, 8]. For some data-intensive applications, energy consumption due to memory access can be more than 50% [15]. In this paper, we focus on the problem of allocating variables in a program between registers and main memory to achieve both short execution time and low energy consumption.

Register allocation for achieving the optimal execution time of a program on a given system is a well known problem and has been studied extensively. In general, existing results can be divided into two categories, local register allocation [5, 14, 17] and global register allocation [2, 4, 11, 13, 21, 22, 25, 26, 27]. A local register allocator considers the code in one basic block (a sequence of code without any branch) at a time and finds the best allocation of the variables in that basic block to achieve the minimum execution time. A global allocator deals with code containing branches, loops, and procedure calls. The global register allocation problems are in general NP-hard [21] and can be modeled as a graph coloring problem [1, 4, 6, 7, 11]. Integer
linear programming approaches and heuristics have been proposed to solve the global register allocation problems [21, 13].

However, optimal-time register allocations do not necessarily consume the least amount of energy [8, 18, 19]. Some researchers have recognized this challenge and have proposed interesting approaches to deal with the optimal-energy register allocation problem [9, 12, 24]. Chang and Pedram [9] gave an approach to solve the register assignment problem in the behavioral synthesis process. They formulated the register assignment problem for minimum energy consumption as a minimum cost clique covering problem and used max-cost flow algorithm to solve it. But memory allocation is not considered in [9]. Gebotys [12] modeled the problem of simultaneous memory partitioning and register allocation as a network flow problem and applied an efficient network flow algorithm to find the best allocation of variables in a basic block to registers and memory. However, the work in [12] did not show how to generalize its approach beyond basic blocks.

In this paper, we extend the work in [12] by investigating the problem of register and memory allocation for low energy beyond basic blocks. In particular, we consider the allocation of variables in a program with branches. Branches are an important structure due to the facts that almost 15% of instructions in a program are branches and that loops and procedure calls can be considered as special branches with more features than common branches [16]. Our approach avoids excessive computational cost. This is achieved by performing the allocation block by block. By carefully analyzing boundary conditions between consecutive blocks, we are able to allocate for each block based on the allocation result from its previous block. In this way, we maintain the global information and make allocation decision to benefit the whole program, not just each basic block. Our approach can be applied to different energy models and variable access models.

Since our register allocator considers allocation beyond basic blocks, it yields better allocations than simply applying the approach in [12] separately to each individual basic block. Furthermore, by carrying out the allocation block by block, our approach avoids the excessive computational cost induced by known methods such as graph coloring and integer linear programming in global register allocation [6, 7, 11, 13, 21]. The time complexity of our algorithm is $O(b \times l(n, k))$, where $b$ is the total number of blocks and $l(n, k)$ is the time complexity for register allocation in a single block that has $n$ variables and $k$ available registers. The function $l(n, k)$ is either $O(n \log n)$, $O(kn \log n)$, or $O(kn^2)$, depending on the models used for register allocation [5, 12]. With today’s IC technology, an optimal energy solution for register allocation will also give an optimal time solution, but the reverse is not necessarily true. We will illustrate the relationship between optimal energy allocation and optimal time allocation in Section 5.

The rest of the paper is organized as follows. Section 2 reviews the energy models, execution models and known approaches for local register allocation for low energy. Section 3 presents our algorithm for low energy register and memory allocation beyond basic blocks. Section 4 gives some examples to illustrate how our algorithm works and to compare our allocation results with other methods. Section 5 discusses in more detail the relation between optimal-time and optimal-energy
results.

2 Preliminaries

Allocating variables to different levels of memory components is usually done by a compiler. Many factors can affect the allocation results, such as the instruction execution order. There are a number of techniques that can be applied to reduce the number of cycles needed to execute a program, including reordering of instructions and loop unrolling. In this paper, we assume that such optimization techniques have been applied to the program and a satisfactory schedule of the program is already given.

Let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of variables in the program under consideration. Given a schedule, the lifetime of a variable $v$ in a basic block is well defined and is denoted by its starting time, $t_s(v)$, and finishing time, $t_f(v)$. We use $V_R$ (resp. $V_M$) to represent the set of variables assigned to the $k$ available registers (resp. memory). Furthermore, let $e_{rv}^M$ (resp. $e_{wv}^M$) be the energy consumed by reading (resp. writing) variable $v$ from (resp. to) memory and $e_{rv}^R$ (resp. $e_{wv}^R$) be the energy consumed by reading (resp. writing) variable $v$ from (resp. to) registers. Denote the total energy consumed by a given program as $E$. Then the objective of the register allocation problem is to map each variable $v \in V$ to register files or memory locations in order to achieve the least energy consumption due to data accesses. That is to minimize Equation (1) below.

$$E = \sum_{v \in V_M} (e_{rv}^M + e_{wv}^M) + \sum_{v \in V_R} (e_{rv}^R + e_{wv}^R)$$

(1)

Different energy models and different data access models have been proposed in the literatures [5, 8, 9, 12, 19]. Depending on the particular models used, the calculation of energy consumption varies. We briefly summarize the models considered in this paper.

The SE models

The static energy model SE assumes that references to the same storage component consume the same amount of energy [8]. That is, $e_{v_i}^M = e_{v_j}^M$ for any $v_i, v_j \in V_M$ and $e_{v_i}^R = e_{v_j}^R$ for any $v_i, v_j \in V_R$. Under this model, the energy consumed by each type access and total number of accesses of each type determine the total energy consumption of a program, which can be expressed as

$$E = N_r^M e_r^M + N_w^M e_w^M + N_r^R e_r^R + N_w^R e_w^R$$

(2)

where $N_r^M, N_w^M, N_r^R,$ and $N_w^R$ represent the total number of memory read, memory write, register read and register write, respectively. Whether a data variable is read only once (SR) or multiple times (MR) after it is defined (written) can have a significant impact on the overall energy estimation. In the following, we discuss briefly the known approaches in minimizing the energy consumption for static energy model under single read and multiple read assumptions.
For the static energy and single read (SE-SR) model, every variable in a basic block is considered to be read only once after it is written and only the total number of accesses of each type and energy consumed by each access type determine the energy consumption of the program. The objective of low energy register allocation becomes minimizing Equation (3) below:

\[ E = N^M (e^M_e + e^M_w) + N^R (e^R_e + e^R_w) \]  

(3)

where \( N^M \) and \( N^R \) represent the total number of references to memory and registers, respectively. Gebotys [12] proposed an approach which transforms the problem to an instance of the network flow problem. Applying the best known network flow algorithm [23], the problem can be solved in \( O(kn^2) \) time. However, we should mention that a more efficient algorithm can be obtained. Considering the facts that \( (e^R_e + e^R_w) \) is less than \( (e^M_e + e^M_w) \) and that the total number of variables in a basic block is fixed, one can see that minimizing Equation (3) is equivalent to minimize \( N^M \). This objective is the same as the one in [5] for optimal-time register allocation. Hence under the SE-SR model, the optimal-time solution is equivalent to the optimal-energy solution.

Carlisle and Lloyd [5] presented a greedy algorithm for putting the maximum number of variables into registers to minimize the execution time of a basic block. With the finishing time \((t_f(v))\) of variables in the basic block sorted in the increasing order, which dominates the time complexity, the algorithm assigns variables to registers using the “best fit” principle. It takes \( O(n \log n) \) time and is easy to implement.

For the multiple read case, let \( h_v \) be the number of times that a variable \( v \) is read, the energy consumption of a program can be expressed as

\[ E = \sum_{v \in V_M} (h_v e^M_{rv} + e^M_w) + \sum_{v \in V_R} (h_v e^R_{rv} + e^R_w) \]  

(4)

The objective is to minimize the total numbers of memory references. It is not difficult to see that optimal-energy solution for the SE-MR model is also equivalent to the optimal-time solution. The algorithm proposed in [12] can be applied to this model. However, the graph model used in [12] can have \( O(n^2) \) edges in the worst case. A better graph model is given in [5], which has only \( O(n) \) edges. Applying the minimum cost network flow algorithm, the optimal-time solution, which is also the optimal-energy solution, can be obtained in \( O(kn \log n) \) time [5] rather than \( O(kn^2) \) as in [12].

In some designs, it is desirable to allow a variable to be assigned to registers in one time interval and switched it to memory in another time interval, in order to minimize the total number of accesses to memory. This is called split lifetime [11]. If the split lifetime model is used, the graph model will need to accommodate all cases where a split occurs. Hence, \( O(n^2) \) graph edges will be needed, and the algorithm in [5] has an \( O(kn^2) \) time complexity, the same as that in [12].

The AE models

In [9, 19], the authors proposed a more sophisticated energy model, called activity based energy model AE, capturing the actual data configuration in a storage. Under this model, the energy
consumed by a program is related to the switched capacitance of successive accesses of data variables which share the same locations. The switched capacitance varies for different access sequences. In particular, the product of the Hamming distance, the number of bits two data items differ [10], or other measurements [9] and the total capacitance of a storage is used to represent the switched capacitance for two consecutive references. We will only consider the activity model for register file access to simplify the problem formulation. (Adopting the activity based model for memory is a simple extension to the algorithm discussed later in this paper.)

Under this model, the total energy consumption of a program can be calculated as

\[ E = N_r^M e_r^M + N_w^M e_w^M + \sum_{v_i \rightarrow v_j, v_i, v_j \in V_R} H(v_i, v_j) C_{RW} V^2 \]

(5)

where \( H(v_i, v_j) \) is the Hamming distance between variables \( v_i \) and \( v_j \), \( v_i \rightarrow v_j \) designates that \( v_j \) is accessed immediately after \( v_i \) and that \( v_i \) and \( v_j \) share the same register, \( C_{RW} \) is the average switched capacitance for register file by read and write operations and \( V_R \) is the operational voltage of the register file.

The objective for optimal energy allocation is to minimize Equation (5). For both the single read and multiple read models, Gebotys [12] formulated the problem as a minimum cost network flow problem with a solution of \( O(kn^2) \) time complexity. The cost for each edge in the network flow problem for AE-SR is different from that in AE-MR, reflecting different read models.

As one can see, different models result in different computational overhead in computing energy consumption. For example, the AE-MR model is the most comprehensive and also most computationally expensive one. Depending on the availability of the models, the architecture features of a system, and the computational overhead that one is willing to incur, any of the above models may be used in energy consumption estimation. Therefore, we consider all the two energy models and read access models in our paper.

The results we discussed above only apply to variables in a basic block, i.e. a piece of straight-line code. Programs from real applications unavoidably contain conditional branches, loops and procedure calls. As we pointed out earlier, the underlying structures of these program control flows are branches. To consider register allocation in a program with branches, one must deal with those variables whose lifetimes extend beyond one basic block. A straightforward way of applying the existing results to such programs would be to enumerate all possible execution paths and treat each path as a straight line code. Two problems arise with this approach. First, the number of paths increases exponentially in the worst case as the number of basic blocks increases. Secondly, each variable may have different lifetimes in different paths. Hence such an approach would be computationally costly and be difficult to obtain an optimal allocation. On the other hand, finding an optimal allocation for all the variables in a general program has been known to be computational intractable (an NP-hard problem).

In the following sections, we discuss an efficient heuristic which can find a nearly optimal solution for low energy consumption register allocation.
3 Register allocation for branches

To avoid the problems encountered by the known approaches discussed in the above section, we propose an iterative approach to handle programs containing branches. A program containing branches is modeled by a tree $T$ in which each node represents a basic block. Denote a node in the tree $T$ by $B$, its parent by $B_p$, and one of its children by $B_c$. We solve the register allocation problem for each basic block in the depth-first search order. The allocation of variables within a block is based on the result from its parent block. That is, a variable in the current block has an initial assignment if it is also alive in the parent block. Any assignment of this variable in the current block must consider the effect of its initial assignment. Furthermore, we do not allow the allocation result in a block to affect the result in its parent block. By allowing the allocation results to propagate down the tree $T$ without back-tracking, we eliminate the excessive computational cost yet obtain superior allocations. A main challenge is how to handle those variables whose lifetimes extend beyond a basic block such that the overall energy consumption of the program is minimized. We have made several observations to help deal with such variables. The key idea is to set up appropriate boundary conditions between parent and child basic blocks and use these conditions to guide the register allocation procedure within the child basic block.

In the following, we discuss in detail our algorithms for the two different energy models.

3.1 Beyond boundary allocation for the static energy model

We first consider the case in which only single read is allowed, and then extend the result to the multiple read case.

For a variable, $v$, if it is defined in a block $B'$ and is also read in another block $B$, the lifetime of $v$ crosses the boundary of the two blocks. The block $B'$ is the parent block, $B_p$, of the block $B$. We say $v$ is in both $B_p$ and $B$. When our algorithm computes the allocation for $B_p$, it performs the computation as if $v$ did not cross the boundary. When we come to the block $B$, we will make the best allocation decision for $v$ in $B$ based on the allocation result of $v$ in $B_p$.

There are totally four different combinations of allocation results for $v$ in the two consecutive blocks: (1) $v$ in register in both $B_p$ and $B$ (R$\rightarrow$R), (2) $v$ in register in $B_p$ but in memory in $B$ (R$\rightarrow$M), (3) $v$ in memory in $B_p$ but in register in $B$ (M$\rightarrow$R), and (4) $v$ in memory both in $B_p$ and $B$ (M$\rightarrow$M), as shown in Figure 1. In Figure 1, solid line segments mean the variable is in register, while dashed segments mean the variable is in memory. (We will use this convention throughout the paper unless stated otherwise explicitly.)

We will analyze the energy consumption for all the four cases, R$\rightarrow$R, R$\rightarrow$M, M$\rightarrow$R, and M$\rightarrow$M. For the R$\rightarrow$R case, the energy consumed by $v$ in block $B_p$ is $e^R_w$ when it is defined, in block $B$ is $e^R_w$ when it is read from the same register, and totally the energy consumed by $v$ in the two consecutive blocks, $B_p + B$, is $e^R_w + e^R_w$. By applying the same analysis to other three cases, we obtain the amount of energy consumed by a crossing-boundary variable in different blocks, as
shown in the column R→R, R→M, M→R, and M→M in Table 1.

For a local variable of B, which is written and read only in block B, the energy consumed by it in B is $e^w_w + e^r_r$ or $e^w_M + e^r_M$ if it is assigned to a register (Local R) or a memory (Local M) location. It consumes no energy in $B_p$ since it does not exist in $B_p$. The energy consumed by a local variable in B is shown in the last two columns (Local R and Local M) in Table 1.

From Table 1, it is easy to see that, based on the allocation result for $B_p$, the best choice for a global variable in B should be:

**M→M :** If v is in memory in $B_p$, it should stay in memory in B to achieve the lowest energy.

Comparing the two columns corresponding to the situation when v is in memory in $B_p$, M→R and M→M, it is clear that if v is in memory in $B_p$, assigned it to a register will cost more energy than assign it to memory in B.

**R→Local :** If v is in register in $B_p$, it should be treated as a brand new local variable in block B to achieve the optimal energy. The energy data for block B and $B_p + B$ in the columns for R→R and Local R is same, while the data differs only by $e^w_w + e^r_r$ in the columns for R→M and Local M in which the total amount of energy is much larger than the difference. So a simple way is to just treat this kind of global variable as a brand new local variable whose allocation is to be determined in block B. If it turns out to be assigned to register in B too, it should stay in the same register as it uses in $B_p$.
Table 2: Weight for global variables and local variables in $B$

<table>
<thead>
<tr>
<th>Block</th>
<th>Global $v$ in Register in $B_p$</th>
<th>Global $v$ in Memory in $B_p$</th>
<th>Local $v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>$e^R_r + e^M_w + h_v e^R_r - h_v e^R_r$</td>
<td>$h_v e^M_w = (e^M_w + e^M_w + h_v e^R_r)$</td>
<td>$h_v e^M_w + e^M_w - (h_v e^R_r + e^R_r)$</td>
</tr>
</tbody>
</table>

Treating $v$, which extends beyond boundary and is assigned to a register in $B_p$, as a brand new variable in $B$ is clearly an approximation. Another way is to assign the actual energy saving, the energy saved by assigning a variable to register comparing with assigning it to memory, as the weight of the variable and construct a network flow and then use minimum-cost network flow algorithms to solve the problem as described in [5] for the weighted case.

A variable, $v$, can also be defined in $B$, not read in the child of $B$, but read in the grandchild or even grand-grandchild of $B$. The analysis for this kind of variable is as same as the above analysis for the variable that defined in $B_p$ and used in $B$, and same conclusion is drawn for the allocation in the children blocks of $B$.

With these rules, we can execute the local register allocation algorithm on each of the blocks in a program. For the simple case, the time complexity of the allocation for the whole program is $O(b n \log n)$, where $n$ is the largest number of local variables in a basic block. For the more complex case, the time complexity is $O(b n \log n)$. The algorithm is shown in Algorithm 1.

As we discussed earlier in Section 2, a variable in a program may be read more than once after it is written. In this case, we can use the same graph model as in [5] but modify the weights associated with certain graph edges. Specifically, we assign the energy saving by assigning a variable to a register as the weight for this variable. For the current block, $B$, the weight of a variable defined in $B$ is $h_v e^M_w + e^M_w - (h_v e^R_r + e^R_r)$, where $h_v$ is the total number of read accesses to $v$ as defined in Section 2. If a variable is alive in the parent block, $B_p$, and is assigned to a register in $B_p$, the weight is for it in the current block $B$ is $h_v e^M_w + e^M_w + e^R_r - (h_v e^R_r)$. Otherwise, if it is allocated to memory in $B_p$, the weight is $h_v e^M_w - (e^M_w + e^R_r + h_v e^R_r)$. Note that the weight is the energy saving when $v$ is allocated to a register in the current block $B$. Since the total energy of accessing all the variables from memory is fixed for a basic block, the more energy saved by allocating variables to registers, the less the energy the block consumes. The weight assignment is summarized in Table 2.

By applying the network flow approach in [5] for weighted intervals on each basic block one after another, we can get a low energy allocation solution for the whole program in $O(b n \log n)$ time.

If splitting is allowed, the graph model in [5], which only allows the register transferred from $v_i$ to $v_j$ if $t_f(v_i) < t_i(v_j)$, is no longer sufficient. In this case, we can use the graph model (to be described in the next section) and the network flow approach presented in [12]. That is, we assign weights as described in Table 2 and use a minimum-cost network flow algorithm to solve the problem in $O(b n^2)$ time.
Algorithm 1 Algorithm for low energy register allocation in static energy model

Input: A scheduled program with blocks, $P = \{B_1, B_2, \ldots, B_m\}$.
Output: The register assignment for every variable $v$ in program $P$.

Definitions:
$B_i(v)$: allocation result for variable $v$ in block $B_i$. $B_i(v)$ is 1 if $v$ is in register, 0 if $v$ is in memory.
$weight(v)$: the weight of variable $v$.

apply algorithm for the unweighted case in [5] to $B_1$
for $i = 2$ to $m$ do
  if using the simple way for approximation then
    for all variables, $v$, alive in $B_{ip}$ and $B_i$ do
      if $B_{ip}(v) = 0$ then
        $B_i(v) = 0$
      else
        $v$ is treated the same as other local variables
      end if
    end for
  else
    for all variables, $v$, alive in $B_{ip}$ and $B_i$ and is read in $B_i$ do
      if $B_{ip}(v) = 0$ then
        $weight(v) = e^M_r - (e^M_r + e^R_w + e^R_r)$
      else
        $weight(v) = e^R_r + e^M_w + e^M_r - (e^R_r)$
      end if
    end for
    for all variables, $v$, alive in $B_{ip}$ and $B_i$ but not read in $B_i$ do
      if $B_{ip}(v) = 0$ then
        $weight(v) = -(e^M_r + e^R_w)$
      else
        $weight(v) = e^R_r + e^M_w$
      end if
    end for
    for all variables written and read in $B_i$ do
      $weight(v) = e^M_w + e^R_M - (e^R_r + e^R_w)$
    end for
    for all variables written but not read in $B_i$ do
      $weight(v) = e^R_w - e^R_r$
    end for
    apply algorithm for the weighted case in [5] to $B_i$
  end if
end for
3.2 Beyond boundary allocation for the activity-based energy model

We first describe our graph model, which is an extension to the model introduced in [12].

Consider a basic block $B$ in $T$. A directed graph $G = (N, A)$ is associated with $B$, which is a generalization of an interval graph. In $G$, there is an arc, $a(v)$, for each data variable $v$ whose lifetime overlaps with the execution duration of $B$. (Note that $v$ can be either a variable referenced in $B$ or a variable which will be referenced by $B$’s descendents.) We denote the two end nodes of $a(v)$ by $n_s(v)$ and $n_f(v)$, which correspond to the starting time $t_s(v)$ (when $v$ is first written) and the finishing time $t_f(v)$ (when $v$ is last read), respectively. Let a critical set, $C_i$, be a set of variables with overlapping lifetimes to one another such that the number of variables in the set is greater than the number $k$ of available registers. For each pair of $C_i$ and $C_{i+1}$ (the index ordering is based on scanning the variables from the beginning to the end of block $B$), let $D_i$ be the set of variables whose lifetimes are in between the minimum finishing time of all variables in $C_i$ and the maximum start time of all variables in $C_{i+1}$. Note that $D_0$ contains the variables whose lifetimes are in between the start time of $B$ and the maximum start time of all variables in $C_1$, and that $D_g$ is the set of variables whose lifetimes are in between the minimum finish time of all variables in $C_g$ (the last critical set in $B$) and the finish time of $B$. Now, an arc is introduced from $n_f(u)$ for each $u \in (C_i \cup D_i)$ to $n_s(v)$ for each $v \in (D_i \cup C_{i+1})$ such that $t_f(u) < t_s(v)$. Intuitively, these arcs represent allowable register sharing among subsequent data reference. In [12], a source node, $S$, and a finish node, $F$, are introduced at the beginning and end of $B$, respectively. Arcs are used to connect $S$ to $n_s(v)$ for each $v \in (D_0 \cup C_1)$ and connect $n_f(u)$ for each $u \in (C_g \cup D_g)$ to $F$. The nodes $S$ and $F$ can be considered as the registers available at the beginning and end of block $B$. An example graph is shown in Figure 2, where the solid lines correspond to the arcs associated with variables in $B$ and the dashed lines are the arcs introduced based on $C_i$’s and $D_i$’s.

To handle the allocation beyond a block boundary, our approach is to decide the register allocation for block $B$ based on the allocation result from $B$’s parent block, $B_p$. Depending on which variable is assigned to which register prior to entering $B$, the amount of energy consumption by each variable in $B$ can be different. Therefore, simply using a single source $S$ to represent the registers available at the beginning of $B$ is no longer sufficient. We generalize the construction of graph $G$ for $B$, where $B$ has both a parent and at least one child, as follows. (The graphs for the root and leaf blocks in $T$ are simply special cases of the graph we discuss.)

For those variables that are referenced only in $B$, we introduce arcs and nodes associated with them in the same way as we discussed above. The start and finish nodes, $S$ and $F$, for $B$ are also maintained. Let the starting and finishing times of block $B$ be $t_s(B)$ and $t_f(B)$, respectively. For each variable $v$ in $B$ whose starting (resp. finishing) time is earlier (resp. later) than the starting (resp. finishing) time of $B$, i.e., $t_s(v) < t_s(B)$ (resp. $t_f(v) > t_f(B)$), we still use two end nodes $n_s(v)$ and $n_f(v)$ in $G$ for the associated arc $a(v)$ (which means that $v$ is considered by all the graphs corresponding to the blocks with which the lifetime of $v$ overlaps). The arcs between these nodes and $S$ or $F$ are defined in the same ways as discussed in the previous paragraphs.
Furthermore, we introduce a register set, $V_{RB}$, which contains the variables residing in registers at the completion of the allocation process for block $B$. Note that $V_{RB}$ becomes uniquely defined after the allocation process of $B$ is completed, and that the size of $V_{RB}$, $|V_{RB}|$, is always less than or equal to $k$, the number of available registers. We expand $G$ by adding a node $n_p(v)$ for each $v \in V_{RB_p}$, where $B_p$ is the parent block of $B$. It is not difficult to see that the variables in $V_{RB_p}$ are the only variables which have a chance to pass on their register locations to variables in $B$ directly. Now, we insert an arc from each $n_p(u)$ to $n_p(v)$ for each $v \in (D_0 \cup C_1)$ (where $D_0$ and $C_1$ for block $B$ are as defined in the previous paragraphs). Comparing our generalized graph with the original graph, one can see that at most $k$ additional nodes and $k \cdot |D_0 \cup C_1|$ additional arcs are used in the generalized graph. Figure 3 shows an example graph $G$ for the current block $B$ assuming that there are three available registers.

Sometimes, a program may read a variable, $v$, more than once. In this case, we introduce an additional node $n_{r_f}(v)$ for each read of $v$ except the last read. Additional arcs are also introduced to model possible register sharing between variables. Due to the page limit, we omit the discussion on this part.

Given the directed graph $G$ for $B$, we are ready to construct the network flow problem associated with $G$. Let $x(n_f(u), n_s(v))$ and $c(n_f(u), n_s(v))$ be the amount of flow and the cost of one unit of flow on arc $a(n_f(u), n_s(v))$, respectively. Denote the total amount of energy consumed by $B$ as $E$. The objective function of our network flow problem can be written as:

\[
\text{Minimize: } E = \sum_{v \in B} (e_{ru}^M + e_{uw}^M) - \sum_{v \in B} e_{uw}^M - \sum_{v \in B} e_{ru}^M
\]
\[
- \sum_{a(p,q) \in A} c(p,q) \cdot x(p,q)
\]  
(6)

where \( A \) is the set of arcs in \( G \). In (6), the first three terms are the amount of energy consumed by \( B \) if all the variables are assigned to memory, and the last term represents the energy saved by allocating certain variables to registers. 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. The values of \( c(p,q) \) are dependent on the types of arcs associated with them, and can be categorized into the following cases.

1) For an arc from a node of type \( n_f \) to another node of type \( n_s \), i.e. \( a(n_f(u), n_s(v)) \), the cost associated to the arc, \( c(n_f(u), n_s(v)) \), is computed by

\[
c(n_f(u), n_s(v)) = e_w^M + e_r^M - \overline{H}(u,v)C_{rw}^R \gamma^2 \quad \forall u, v \in N
\]  
(7)

where \( N \) is the set of nodes in \( G \). This is the amount of energy saved by reading \( u \) from a register and writing \( v \) to the same register.

2) For an arc from a node of type \( n_p \) to another node of type \( n_s \), i.e. \( a(n_p(u), n_s(v)) \), the cost associated to the arc, \( c(n_p(u), n_s(v)) \), is defined differently. There are a total of 7 cases to be considered.

2.1) If \( u \) is not in \( B \) (i.e., \( u \)'s lifetime does not overlap with that of \( B \)), and \( v \) is written in \( B \), the cost \( c(n_p(u), n_s(v)) \) is computed by

\[
c(n_p(u), n_s(v)) = e_w^M - \overline{H}(u,v)C_{rw}^R \gamma^2
\]  
(8)
2.2) If \( u \) is not in \( B \), and \( v \) has been assigned to a register during the allocation process of \( B_p \),
\[
c(n_p(u), n_s(v)) = -\overline{\mathcal{H}}(u, v) C_{rw}^{R} \gamma^2,
\]
(9)

2.3) If \( u \) is not in \( B \), and \( v \) has been assigned to memory during the allocation process of \( B_p \),
\[
c(n_p(u), n_p(v)) = -e_r^M - \overline{\mathcal{H}}(u, v) C_{rw}^{R} \gamma^2,
\]
(10)

2.4) If \( u \) is in \( B \), and \( v \) is written in \( B \), the cost \( c(n_p(u), n_s(v)) \) is the same as defined in (9) for Case 2.2.

2.5) If \( u \) is in \( B \), and \( v \) has been assigned to a register during the allocation process of \( B_p \),
\[
c(n_p(u), n_s(v)) = -e_w^M - \overline{\mathcal{H}}(u, v) C_{rw}^{R} \gamma^2,
\]
(11)

2.6) If \( u \) is in \( B \), and \( v \) has been assigned to memory during the allocation process of \( B_p \),
\[
c(n_p(u), n_p(v)) = -e_w^M - e_r^M - \overline{\mathcal{H}}(u, v) C_{rw}^{R} \gamma^2,
\]
(12)

2.7) If \( u \) and \( v \) represent the same variable, the cost \( c(n_p(u), n_s(v)) \) is simply assigned to zero.

3) For an arc from start node \( S \) to another node of type \( n_s \), i.e. \( a(S, n_s(v)) \) for \( v \in D_0 \cup C_1 \), we need to have three different cost functions.

3.1) If \( v \) is written in \( B \),
\[
c(S, n_s(v)) = e_w^M - \overline{\mathcal{H}}(0, v) C_{rw}^{R} \gamma^2
\]
(13)

where \( \overline{\mathcal{H}}(0, v) \) is the average Hamming distance between 0 and a variable \( v \), and is normally assumed to be 0.5.

3.2) If \( v \) has been assigned to a register during the allocation process of \( B_p \),
\[
c(S, n_s(v)) = -\overline{\mathcal{H}}(0, v) C_{rw}^{R} \gamma^2
\]
(14)

3.3) If \( v \) has been assigned to memory during the allocation process of \( B_p \),
\[
c(S, n_s(v)) = -e_w^M - \overline{\mathcal{H}}(0, v) C_{rw}^{R} \gamma^2
\]
(15)

4) For an arc from a node of type \( n_f \) to the finish node, \( F \), i.e. \( a(n_f(v), F) \) for \( v \in D_g \cup C_g \), we need to have two different cost functions.

4.1) If \( v \) is read in \( B \),
\[
c(n_f(v), F) = e_r^M
\]
(16)

4.2) If \( v \) is not read in \( B \), the cost \( c(n_f(v), F) \) is simply assigned to zero.
5) For an arc from a node of type \( n_s \) to another node of type \( n_f \), which is the arc corresponding to the same variable, the cost associated to the arc is assigned to zero.

Using the above equations, the objective function for the network flow problem will be uniquely defined. The constraints for the network flow problem is defined based on the number of registers available to the arc. They are summarized as following:

\[
\sum_{v \in (C_p \cup D_p)} x(n_f(v), F) \leq k
\]  

(17)

\[
\sum_{v \in (D_b \cup C_1)} x(S, n_s(v)) \leq \max\{0, k - |V_{RB_p}|\}
\]  

(18)

\[
\sum_{v \in (D_b \cup C_1)} x(S, n_s(v)) + \sum_{v \in (D_b \cup C_1)} x(n_p(u), n_s(v)) = \sum_{v \in (C_p \cup D_p)} x(n_f(v), F)
\]  

(19)

\[
\sum_{p} x(p, z) = \sum_{q} x(z, q) \quad \forall z \in N
\]  

(20)

\[
x(p, q) \leq 1 \quad \forall a(p, q) \in A
\]  

(21)

Applying a network flow algorithm such as the one in [20] to our network flow problem instance, we can obtain the value of each \( x(p, q) \) in \( O(kn^2) \) time for the block \( B \), where \( k \) is the number of available registers and \( n \) is the number of variables whose lifetimes overlap with the lifetime of \( B \). If the resulted \( x \) value of the arc associated with a variable in \( B \) 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 tree in the depth-first search order as we discussed at the beginning of this section.

Here we should point out that our method can also be used if one wishes to explore program execution paths to determine the register allocation. For example, we can associate each execution path of a program with a priority. One way to assign priorities is based on the execution frequency of each path (obtained from profiling). Another way is based on the timing requirements. To solve the register allocation problem, one can start with the highest priority path, \( P_1 \), and proceed to the lower priority ones sequentially. For the first path, form a super block for the path and use the local register allocation algorithm in [12] to find the best allocation for this single path. For the next path, remove the basic blocks on this path whose register assignments have been determined. Then form a super block \( B' \) for the rest of the basic blocks in this path. To construct the graph for this super block \( B' \), we need to consider all those \( B_p \)'s that have a child in \( B' \) and introduce the necessary \( n_p \) nodes to complete the graph. The network flow problem for the super block \( B' \) can use the same formulations as we discussed above. The process is repeated for all subsequent paths. By applying our algorithm, the variable allocation decisions made for the higher priority paths will not be changed by the allocation processes of the lower priority paths. Rather, the results from the higher priority paths are used to insure that good allocations are found for the lower priority paths based on these results. Hence, we have effectively eliminated the conflicting assignments that can arise when solving the allocation problem for each path separately.
3.3 Optimizing energy implies optimizing time

Here we want to give a discussion about the relation between optimal-time register allocation and optimal-energy allocation. As stated earlier in this paper, based on the static energy model, the optimal-time allocation is equivalent to the optimal-energy allocation. That is why we can use the optimal-time allocation algorithm in [5] for our optimal-energy task in a basic block.

When activity based energy model is considered, the energy consumed by a basic block is not only related to the total number of references to memory and register file but also related to which variables share same registers and memory locations. But the execution time is only related to the total number of references to register and memory when each reference to memory takes same clock cycles, and so does each register reference. In this paper, we only consider the reference sequence in register access and assume that each memory access consumes same energy. Under this assumption, we want to find out whether a primal-energy allocation gives the optimal-time solution or not.

Let’s consider two allocators. Allocator A1 optimizes time then choose one allocation result with minimum energy as its result for low energy allocation and A2 just optimizes for energy. If the total memory references by the two allocators are different, then A2 allocates more variable to memory but somehow by managing the sharing of registers between variables smartly, it achieves optimal energy.

The energy consumed of the two allocation results would be:

\[ E_1 = 2N_1e^M + (N - N_1)H_1e^R \]  \hspace{1cm} (22)

\[ E_2 = 2N_2e^M + (N - N_2)H_2e^R \]  \hspace{1cm} (23)

Where \( E_1 \) and \( E_2 \) are the energy consumed by all the variables, \( N_1 \) and \( N_2 \) are the number of variables allocated to memory, \( H_1 \) and \( H_2 \) are the average Hamming distance for all the variables in registers, \( e^M \) and \( e^R \) are the energy consumed by one memory reference and register reference, respectively.

By assumption, we have \( E_2 < E_1 \) and \( N_2 > N_1 \).

Using Equation (23) - (22), we get

\[ 2(N_2 - N_1)e^M < (N - N_2)(H_1 - H_2)e^R + (N_2 - N_1)H_1e^R \]  \hspace{1cm} (24)

That is

\[
\frac{e^M}{e^R} < \frac{(N - N_2)(H_1 - H_2)}{2(N_2 - N_1)} + H_1
\]  \hspace{1cm} (25)

Since in a program, almost 15% of the instructions are branches [16], each block will not be very big. The average Hamming distance would be around 0.5 between two variables. In current system, the energy consumed by one memory reference is usually much more than a register reference[3]. These facts together make the condition shown in Equation (25) difficult to be true.
We will compare the result of our algorithm to the result of the stack model. The example procedure is simple. The example procedure has three possible results and two possible outcomes.

Figure 4 shows a graph of a program with two branches. Each line segment in the graph corresponds to one branch of the program.

We will use the following definition of a program as an example and apply our algorithm to it.

**Example**

The solution is also the solution for optimal-time for most of today's systems. In Section 3, this is under the stack model. The optimal-time model is very similar to the stack model. The optimal-time model and the stack model are very similar. We now compare the two models.

In Figure 4, the line shows the relation between the stack model and the optimal-time model. As can be seen, the optimal-time model is very similar to the stack model.

In this section, we will use an example to illustrate how our algorithm works for global register allocation.

The figure shows the relation between the stack model and the optimal-time model.

So the total number of variables in memory for optimal-time allocation will most likely be...
of path-optimal approach, which optimize for each path separately, a traditional graph coloring algorithm in [1] and the result from integer linear programming approach in [13].

The straightforward way of doing global register allocation for optimal energy would be enumerate all the possible paths, then treat each path as a single block and apply the local register allocation algorithms for optimal energy to get the allocation for each path. But this will cause two problems as pointed out earlier in this paper. The existence of exponential many possible paths makes the computation cost prohibitively high. And for a variable alive in different paths, when the allocation results conflict with each other, a tradeoff is inevitable.

The result for doing allocation for two possible paths separately, represented by PATH, in this example is shown in Figure 6.

It is obvious that the allocation result for c in B1 in different paths (P1 and P2) is conflicting with each other. Variable c could either stay in register or memory in B1 to resolve the conflicting results from two paths. The energy consumption will be different for different resolving decision.
If a traditional graph coloring, and for all those algorithms based on graph coloring approaches, is used, an interference graph needs to be built before coloring approaches applied on it. The interference graph has $n$ nodes to represent the $n$ variables in the program and an edge between two nodes if the lifetimes of the two corresponding variables overlap [7]. Two nodes can not have same color if there is an edge in-between them. The interference graph for the example is show in Figure 7.

Applying the graph coloring approach in [1] to the interference graph, the result is shown in figure 8.

By our algorithm, we first do the allocation for the first block, $B_1$, using the algorithm from [5]. The allocation result is shown in $B_1$ in Figure 9 and the time complexity is $O(n\log n)$, where $n$ is five for the block $B_1$.

For block $B_2$, variable $c$ comes from $B_2$’s parent block, $B_1$, and is in register in $B_1$, so $c$ is treated as a brand new variable, just as same variables $f$, $g$ and $h$. And by using the local register
allocation algorithm as used for $B_1$, we get the result shown in $B_2$ in Figure 9. For block $B_3$, variable $c$ and $d$ are all from $B_3$’s parent block, $B_1$. In $B_1$, $c$ is in a register while $d$ is in memory. So $c$ is treated as a new variable, as if it were as same as $i$, $j$, $k$, $l$ and $m$. While $d$ is in memory in $B_1$, according to our analysis, it should stay in memory in $B_3$. The allocation result for block $B_3$ is shown in $B_3$ in Figure 9. Thus, we finished the optimal-energy global register allocation for the whole program and the result is shown in Figure 9.

If we apply the integer linear program method in [13] to the example with equal possibility of executing $P_1$ and $P_2$, we get the exactly same result after using a linear programming solver to solve the problem which has more than 30 variables and constrain functions.

Let us compare the energy consumption of our algorithm with other different allocation algorithms, such as enumeration of paths(PATH), including the different decision to resolve the conflicting results, graph coloring approach, and integer linear programming approach. We have mentioned that the integer linear programming method produces the same allocation result with our algorithm. If the resolving decision is to put $c$ in register in $B_1$, the allocation result in Figure 6 will be as same as our result shown in Figure 9. So we just need to compare our allocation result with the other two different results produced by enumerating paths with $c$ in memory in $B_1$ and graph coloring. Assume that path $P_1$ (resp. $P_2$) has 100% (resp. 0%), 80% (resp. 20%), 50% (resp. 50%), 20% (resp. 80%) and 0% (resp. 100%) possibility to be executed, the energy consumption by different allocation approaches for the whole program is shown in Table 3, where $(R_{rw})$ denotes the energy $e_w^R + e_r^R$ for one register read and write, and $(M_{rw})$ denotes the energy $e_w^M + e_r^M$ for one memory read and write.

The difference of the energy consumption between our algorithm and the other two approaches is shown in Table 4.

By considering the fact that memory references consume much more energy than register references, the energy consumption based on our algorithm is less than that based on the exhaustive
approach and graph coloring, except when the program mainly exercises path \( P_2 \) with \( B_1 \) and \( B_3 \). If this is the case, we can use the priority-based approach discussed at the end of the last section to get the same result for the highest priority path \( P_2 \) as other approaches. Hence, our algorithm is quite powerful for solving the register allocation problem for programs with branches.

5 Conclusion

In this paper, we proposed a new algorithm to deal with those variables whose lifetimes extend beyond basic blocks for low energy register allocation. Our algorithm has much less time and space complexity than other approaches in doing register allocation for variables extending beyond basic blocks. The results so far are very promising and the result for the example in Section 4 shows this clearly. More experiments with larger code sizes are being conducted. We are also investigating on how to deal with more complex control flow structures, such as loops and procedure calls.

6 Acknowledgment

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

References


