Efficient 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 the allocation results benefiting the whole program. The computational cost for solving the allocation problem is rather low comparing with known approaches while the quality of the results are very encouraging.

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 [25]. A lot of research has been done in improving the power consumptions of various components in computer systems [8, 13, 19, 20, 25]. 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, 17]. Hence allocation of variables in a program to different storage elements plays an important role toward achieving high performance and low energy consumption [13]. 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%[16]. 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, 15, 18] and global register allocation
[2, 4, 11, 14, 22, 23, 27, 28, 29]. 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 [22] 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 [22, 14].

However, optimal-time register allocations do not necessarily consume the least amount of energy [8, 19, 20]. Some researchers have recognized this challenge and have proposed interesting approaches to deal with the optimal-energy register allocation problem [9, 13, 25]. 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 [13] modeled the problem of simultaneous memory partitioning and register allocation as a network flow problem and an efficient network flow algorithm can be applied to find the best allocation of variables in a basic block to registers and memory. However, the work in [13] is only applicable to basic blocks.

In this paper, we extend the work in [13] by investigating the problem of register and memory allocation for low energy beyond basic blocks. We consider programs that may contain any combination of branches, merges, and loops. The basis of our approach is to perform the allocation block by block. By carefully analyzing boundary conditions between consecutive blocks, we are able to find the best allocation for each block based on the allocation results from previous blocks. 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 and variable access models. 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 different energy and access models used.

The rest of the paper is organized as follows. Section 2 reviews the energy models and known approaches for local register allocation for low energy. We also give an overview of how our allocation approach works. Section 3 gives our approach to do the allocation for the branch control structure. Section 4 presents our algorithm for low energy register allocation for the merge structure in a control flow graph. In section 5, we discuss how to treat loops to get better register allocation results. Section 6 summarizes the experimental results and concludes with some discussions.
2 Overview

Allocating variables to different levels of storage components is usually done by a compiler. Many factors can affect the allocation results. There are a number of techniques that can be applied to reduce the number of cycles needed to execute a program, including instructions reordering 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.

2.1 Energy models and known approaches

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 $e^M_{rv}$ (resp., $e^M_{wv}$) be the energy consumed by reading (resp., writing) variable $v$ from (resp., to) memory and $e^R_{rv}$ (resp., $e^R_{wv}$) be the energy consumed by reading (resp., writing) variable $v$ from (resp., to) registers. If $v$ is not specified, $e^M_r$ (resp., $e^M_w$) represents the energy consumed by one read (resp., write) access to the memory and $e^R_r$ (resp., $e^R_w$) represents the energy consumed by one read (resp., write) access 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

$$E = \sum_{v \in V_M} (e^M_{rv} + e^M_{wv}) + \sum_{v \in V_R} (e^R_{rv} + e^R_{wv})$$

(1)

Different energy models and different data access models have been proposed in the literatures [5, 8, 9, 13, 20]. Depending on the particular models used, the calculation of energy consumption varies. One of the energy model is the static energy model SE, which assumes that references to the same storage component consume the same amount of energy [8]. This model is like the reference-time model, where accesses to the same storage component take the same amount of time, no matter of what is the value of the referenced data. In [9, 20], 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.

Whether a data variable is read only once (SR) or multiple times (MR) after it is defined (written) also has an effect on the overall energy estimation. 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 under the signal read model. Their approach also gives an optimal energy consumption for the static energy model. It takes $O(n \log n)$ time and is quite easy to implement.

For the multiple-read case, the algorithm proposed in [13] uses a graph which 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 [13].

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 the two variable access models in our paper.

For the activity based energy model, we will only consider it 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

$$E = N_r^M e_r^M + N_w^M e_w^M + \sum_{v_i \to v_j, v_i, v_j \in V_R} H(v_i, v_j) C_{rw} V^2$$

(2)

where $H(v_i, v_j)$ is the Hamming distance between $v_i$ and $v_j$, $v_i \to v_j$ 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 reference and $V_R$ is the operational voltage of the register file.

The objective for optimal energy allocation is to minimize the objective function (2). For both the single read and multiple read models, Gebotys [13] 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 different read models is also different.

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 [13].

The results we discussed above only apply to variables in a basic block, i.e., a piece of straight-line code. But real application programs may contain some control structures, such as branches, merges and loops. A merge is the case where one block has more than one parent blocks. Such type of control flows occurs when an instruction has more than one entry point (due to branching, looping, or subroutine calls). To consider register allocation in a program with these control structures, one must deal with those variables whose lifetimes extend beyond one basic block. A straightforward way of applying the existing approaches to such programs would be to enumerate
all possible execution paths and treat each path as a piece of 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).

2.2 Overview of our approach

In the following, we outline our heuristic approach which can find superior solutions for the minimum energy register allocation problem. To avoid the difficulties encountered by the known approaches discussed above, we use an iterative approach to handle programs containing non-straight-line code. Such program is modeled by a directed graph $G$ in which each node represents a basic block. Given the nature of a program, the corresponding graph always has a node whose in-degree is zero. Furthermore, for each loop, the edges that branch backwards can be identified. A topological order for the nodes in $G$ can thus be established after removing those edges branching backwards. To ease our explanation, we generalize the concept of parent and child used in trees. A parent (resp., child) of a block $B$ is a block whose corresponding node in $G$ is an immediate ancestor (resp., descendent) of the node corresponding to $B$. Hence, a block may have multiple parents blocks (in the case of a merge) and it may be a parent block to itself (in the case of a loop).

We solve the register allocation problem for each basic block in the topological order. The allocation of variables within a block is based on the results from its parent blocks. 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 blocks. By allowing the allocation results to propagate “downwards” 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 blocks.

We first describe the graph model used in [13], and then introduce extensions to handle variables beyond basic blocks. Consider a basic block $B$ in $G$. A directed graph $G = (N,A)$, which is a generalization of an interval graph, is associated with $B$. For each data variable $v$ whose lifetime overlaps with the execution duration of $B$, two nodes, $n_s(v)$ and $n_f(v)$ are introduced, 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. Note that $v$ can be either a variable referenced in $B$ or a variable which will be referenced by $B$’s descendents.
Several different types of arcs are used in $G$. First, each pair of $n_s(v)$ and $n_f(v)$ is connected by an arc $a(n_s(v), n_f(v))$. To define the rest of the arcs, we introduce the concept of critical set. A critical set, $C_i$, is a set of variables with overlapping lifetimes to one another such that the number of variables in the set is greater than $k$, the number 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 starting time of all variables in $C_{i+1}$. Note that $D_0$ contains the variables whose lifetimes are in between the starting time of $B$ and the maximum starting time of all variables in $C_1$, and that $D_g$ is the set of variables whose lifetimes are in between the minimum finishing time of all variables in $C_g$ (the last critical set in $B$) and the finishing 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 references.

Similar to [13], 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 to 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 1, where the solid lines correspond to the arcs associated with variables in $B$ and the dash lines are the arcs introduced based on $C_i$’s and $D_i$’s. In this graph, variable $b$, $c$ and $d$ form the first critical set, $C_1$, while variable $a$ and $b$ form the set $D_0$.

To handle the allocation beyond a block boundary, the graph needs to be modified. Substantial modifications to the existing approaches as well as new ideas are proposed here in order to efficiently handle the branch, merge, and loop structures.
3 Register Allocation for the Branch Structure

In this section, we discuss in detail our algorithms of register allocation for the branch structure 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→R), (2) \( v \) in register in \( B_p \) but in memory in \( B \) (R→M), (3) \( v \) in memory in \( B_p \) but in register in \( B \) (M→R), and (4) \( v \) in memory both in \( B_p \) and \( B \) (M→M), as shown in Figure 2. In Figure 2, 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→R, R→M, M→R, and M→M. For the R→R case, the energy consumed by \( v \) in block \( B_p \) is \( e^R_w \) when it is defined, in block \( B \) is \( e^R_r \) when it is read from the same register, and totally the energy consumed by \( v \) in the two consecutive blocks, \( B_p + B \), is \( e^R_w + e^R_r \). By applying the same analysis to other three cases, we obtain the amount of energy consumed by a crossing-boundary variable in different cases, 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
Table 1: Energy consumption in different blocks under different allocation results

<table>
<thead>
<tr>
<th>Block</th>
<th>R→ R</th>
<th>R→ M</th>
<th>M→ R</th>
<th>M→ M</th>
<th>Local R</th>
<th>Local M</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_p</td>
<td>$e_w^R$</td>
<td>$e_w^M + e_R^M$</td>
<td>$e_w^M + e_R^M$</td>
<td>$e_w^M + e_R^M$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>$e_r^R$</td>
<td>$e_R^R$</td>
<td>$e_R^R + e_w^R + e_R^M$</td>
<td>$e_R^R + e_w^R + e_R^M$</td>
<td>$e_r^R + e_w^R + e_R^M$</td>
<td>$e_r^R + e_w^R + e_R^M$</td>
</tr>
<tr>
<td>B_p + B</td>
<td>$e_w^R + e_r^R$</td>
<td>$e_r^R + e_w^R + e_R^M$</td>
<td>$e_w^M + e_R^R + e_w^R + e_R^M$</td>
<td>$e_w^M + e_R^R + e_w^R + e_R^M$</td>
<td>$e_w^M + e_R^R + e_w^R + e_R^M$</td>
<td>$e_w^M + e_R^R + e_w^R + e_R^M$</td>
</tr>
</tbody>
</table>

it in B is $e_w^R + e_r^R$ 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 $e_w^R + e_r^R$ 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.

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 $O(bn\log n)$, where n is the largest number of local variables in a basic block. For the more complex case, the time complexity is $O(bkn\log n)$. 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 written, we can use the same graph model as in [5] but modify the weights
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, $v$, live in $B_{i-1}$ and $B_i$ do
            if $B_{i-1}(v) = 0$ then
                $B_i(v) = 0$
            else
                $v$ is treated the same as other local variables
            end if
        end for
    end if

apply algorithm for the unweighted case in [5] to $B_i$

elseif
    for all variables, $v$, live in $B_{i-1}$ and $B_i$ and is read in $B_i$ do
        if $B_{i-1}(v) = 0$ then
            $weight(v) = c_r^M - (e_r^M + e_w^R + e_r^R)$
        else
            $weight(v) = e_r^R + e_w^M + e_r^M - (e_r^R)$
        end if
    end for

    for all variables, $v$, live in $B_{i-1}$ and $B_i$ but not read in $B_i$ do
        if $B_{i-1}(v) = 0$ then
            $weight(v) = -(e_r^M + e_w^R)$
        else
            $weight(v) = e_r^R + e_w^M$
        end if
    end for

    for all variables written and read in $B_i$ do
        $weight(v) = e_w^M + e_r^M - (e_r^R + e_w^R)$
    end for

    for all variables written but not read in $B_i$ do
        $weight(v) = e_w^M - e_w^R$
    end for

apply algorithm for the weighted case in [5] to $B_i$
end if
end for


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>$e^R_v + e^M_v + h_v e^M_v - h_v e^R_v$</td>
<td>$h_v e^M_v - (e^R_v + e^M_v + e^{R}_v)$</td>
<td>$h_v e^M_v + e^M_v - (h_v e^R_v + e^{R}_v)$</td>
</tr>
</tbody>
</table>

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 $h_v e^M_v + e^M_v - (h_v e^R_v + e^{R}_v)$, where $h_v$ is the total number of read accesses to $v$ as defined in Section 2. If a variable is live 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 $h_v e^M_v + e^M_v + e^{R}_v - (h_v e^R_v)$. Otherwise, if it is allocated to memory in $B_p$, the weight is $h_v e^M_v - (e^M_v + e^{R}_v + h_v e^R_v)$. 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 $O(bkn\log n)$ 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 [13]. That is, we assign weights as described in Table 2 and use a minimum-cost network flow algorithm to solve the problem in $O(bkn^2)$ time.

3.2 Register allocation for branches for the activity-based energy model

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 leave 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
Figure 3: The graph $G$ for block $B$ based on the allocation of its parent block $B_p$.

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 $n_s(v)$ 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 $n_{r_i}(v)$ 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:

Minimize: $E =$
\[
\sum_{v \in B} (e^M_{rv} + e^M_{uw}) - \sum_{v \in B|l_v(B)} e^M_{uv} - \sum_{v \in B|t_f(B)} e^M_{rw} - \sum_{a[p,q] \in A} c(p, q) \cdot x(p, q) \tag{3}
\]

where \(A\) is the set of arcs in \(G\). In (3), 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

\[
c(n_f(u), n_s(v)) = e^M_w + e^M_r - \overline{H}(u, v)C_{rw}^R \gamma^2 \quad \forall u, v \in N \tag{4}
\]

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

\[
c(n_p(u), n_s(v)) = e^M_w - \overline{H}(u, v)C_{rw}^R \gamma^2 \tag{5}
\]

2.2) If \(u\) is not in \(B\), and \(v\) has been assigned to a register during the allocation process of \(B_p\),

\[
c(n_p(u), n_s(v)) = -\overline{H}(u, v)C_{rw}^R \gamma^2, \tag{6}
\]

2.3) If \(u\) is not in \(B\), and \(v\) has been assigned to memory during the allocation process of \(B_p\),

\[
c(n_p(u), n_s(v)) = -e^M_w - \overline{H}(u, v)C_{rw}^R \gamma^2, \tag{7}
\]

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 (6) for Case 2.2.

2.5) If \(u\) is in \(B\), and \(v\) has been assigned to a register during the allocation process of \(B_p\),

\[
c(n_p(u), n_s(v)) = -e^M_w - \overline{H}(u, v)C_{rw}^R \gamma^2, \tag{8}
\]

2.6) If \(u\) is in \(B\), and \(v\) has been assigned to memory during the allocation process of \(B_p\),

\[
c(n_p(u), n_s(v)) = -e^M_w - e^M_r - \overline{H}(u, v)C_{rw}^R \gamma^2, \tag{9}
\]
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 \( B \),
\[
c(S, n_s(v)) = e_w^M - \bar{H}(0, v)C_{rw}V^2
\]

where \( \bar{H}(0, v) \) 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 \( B_p \),
\[
c(S, n_s(v)) = -\bar{H}(0, v)C_{rw}V^2
\]

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

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 \( B \),
\[
c(n_f(v), F) = e_r^M
\]

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:

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

\[
\sum_{v \in (D_0 \cup C_1)} x(S, n_s(v)) \leq \max\{0, k - |V_{RB_p}|\}
\]

\[
\sum_{v \in (D_0 \cup C_1)} x(S, n_s(v)) + \sum_{v \in (D_0 \cup C_1)} x(n_p(u), n_s(v)) = \sum_{v \in (C_g \cup D_g)} x(n_f(v), F)
\]

\[
\sum_p x(p, z) = \sum_q x(z, q) \quad \forall z \in N
\]

\[
0 \leq x(p, q) \leq 1 \quad \forall a(p, q) \in A
\]
Applying a network flow algorithm such as the one in [21] 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, \( P_i \), 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 [13] 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.

4 Register Allocation for Blocks with Merging Parent Blocks

In this section we present our approach to handle the merge case. 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. If a variable, \( v \), is alive in \( B \) but is not defined (written) in \( B \), \( v \) must be alive when leaving every \( B \)'s parents. Since the allocation results for \( v \) in different parent blocks may be different, we cannot simply do the allocation for \( v \) in \( B \) based on the allocation result from one particular parent block. Hence our register allocation approach for programs with branches, where each basic block has only one parent block, is not adequate for a block when more than one blocks merging into it. To handle the control structure of merge blocks, we devise a method to capture the non-unique assignments of variables to registers. Based on this approach, we formulate the problem of register allocation for both the static energy model and the activity based energy model as an instance of the network flow problem.

Let the parent blocks of \( B \) be \( B^{p_1}, B^{p_2}, \ldots, B^{p_m} \), assuming \( m \) is the total number of blocks
merge into $B$ and $m > 1$. Each parent block, $B^p_i$ is associated with a probability, $P(B^p_i)$, indicating how frequently the block is executed. (Such probabilities can be obtained from profiling information.) After the allocation for a parent block is finished, the allocation decision for a variable in $B^p_i$ which goes beyond the boundary of $B^p_i$ and $B$ is fixed. We define the probability of $v$ allocated to a register (resp., memory) in $B^p_i$ as $P(v, R, B^p_i)$ (resp., $P(v, M, B^p_i)$). Furthermore, let $P(v, R_j, B^p_i)$ be the probability of $v$ being assigned to the register $R_j$ in block $B^p_i$. We also define $P(v, R, \sum_{j=1}^m B^p_j)$ (resp., $P(v, M, \sum_{j=1}^m B^p_j)$) as the total probability of $v$ allocated to a register (resp., memory) in all the parent blocks, $P(v, R_j, \sum_{j=1}^m B^p_j)$ as the total probability of $v$ allocated to the register $R_j$ in all the parent blocks. The following relations hold for the variables defined above, where $k$ is the total number of registers.

\[
\begin{align*}
\sum_{j=1}^{j=k} P(v, R_j, B^p_i) &= P(v, R, B^p_i) \\
\sum_{i=1}^{i=m} P(v, R_j, B^p_i) &= P(v, R_j, \sum_{i=1}^m B^p_i) \\
\sum_{i=1}^{i=m} P(v, R, B^p_i) &= P(v, R, \sum_{j=1}^m B^p_j) \\
\sum_{i=1}^{i=m} P(v, M, B^p_i) &= P(v, M, \sum_{j=1}^m B^p_j) \\
\sum_{i=1}^{i=m} P(v, R_i, \sum_{j=1}^m B^p_j) + P(v, M, \sum_{j=1}^m B^p_j) &= \sum_{i=1}^{i=m} P(B^p_i) = P(B)
\end{align*}
\]

We modify the flow graph to be used in the merge case as follows. The source node, $S$, and the finish node, $F$, are kept as the same. The nodes used to represent a variable, $v$, are still the same as $n_s(v)$ for the starting node, and $n_f(v)$ for the finishing node. Because the register assignments in different parent blocks may be different, a cross boundary variable may have different allocation assignment in different parent blocks. To reflect this fact, a critical set, $C_i$, is defined as a set of variables with overlapping lifetimes to one another such that the number of variables written in $B$ in the set is greater than the number $k$ of available registers. Furthermore, we introduce a new set of register nodes, $n_r(i)$, ($i = 1, 2, \ldots, k$), which correspond to the registers. In addition to all the arcs defined in Section 2, we add new arcs from source node, $S$, to each register node, $n_r(i)$. A complete bipartite graph is formed between the register nodes and the starting nodes for all variables in the first critical set $C_1$ or $D_0$. An example flow graph for a block $B$ with multiple parent blocks is shown in Figure 4, assuming that there are two registers, $R_1$ and $R_2$, available in the system.

In minimizing energy consumption by a single block with multiple blocks merging into it, one should minimize the overall energy consumption under all execution scenarios for this block. Thus, the objective function for the flow problem needs to take into account the probabilities of executing different parent blocks. For different energy models, static energy model and activity based energy model, the objective function and the cost associated with arcs are slightly different. To reduce repetition, we will integrate the presentation of the formulations for both models and point out the differences wherever necessary. The objective function can be written as:
Minimize:  \[ E = P(B) \sum_{v \in B} (e_r^M + e_w^M) \]
\[- \sum_{v \in B | t_v < t_w(B)} (P(v, M, \sum_{j=1}^{m} B^{p_j})e_w^M - P(v, R, \sum_{j=1}^{m} B^{p_j})\lambda e_r^R) \]
\[-P(B) \sum_{v \in B | t_v > t_w(B)} e_r^M - P(B) \sum_{a(p, q) \in A} c(p, q) \cdot x(p, q) \quad (19) \]

where
\[ \lambda = \begin{cases} 
1 & \text{if static energy model} \\
0 & \text{otherwise}
\end{cases} \]

The first three terms of computing the objective function (19) are the amount of energy consumed by block \( B \) if all the variables are assigned to memory, while the last term represents the energy saved by allocating certain variables to registers. The difference in the \( \lambda \) value is due to the fact that in the activity based energy model, the energy consumption of 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 (19). 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 \( n_e \), i.e., \( a(n_f(u), n_e(v)) \), the cost associated to the arc, \( c(n_f(u), n_e(v)) \), is computed differently for two different kinds of energy model. For the static energy model,
\[ c(n_f(u), n_e(v)) = e_w^M + e_r^M - (e_w^R + e_r^R) \quad \forall u, v \in N \quad (20) \]
while for the activity based energy model,

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

(21)

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 \( n_r(i) \) to another node of type \( n_s \), i.e., \( a(n_r(i), n_s(v)) \), the cost associated to the arc, \( c(n_r(i), n_s(v)) \), can take one of the following values.

2.1) If \( v \) is written in \( B \), for the static energy model,

\[ c(n_r(i), n_s(v)) = e_w^M - e_r^R \]  

(22)

while for the activity based energy model,

\[ c(n_r(i), n_s(v)) = e_w^M - \sum_{j=1}^{m} \sum_{u \in V_{Rb^j}|P(u, R_i, B^j) \neq 0} P(u, R_i, B^j)\overline{\Pi}(u, v)C_{rw}^R \]  

(23)

For the static energy model, the energy saving is the energy of a memory write, \( e_w^M \), minus the energy of a register write, \( e_r^R \). On the other hand, for the activity based energy model, the energy for writing \( v \) into a register is determined by the hamming distance between \( v \) and the variable \( u \) that occupied \( R_i \) at the entry of block \( B \). Since the probability for variable \( u \) occupying \( R_i \) is \( P(u, R_i, B^j) \), the average hamming distance would be the summation of the hamming distance between \( v \) and all the \( u \) which has non-zero probability of occupying \( R_i \) in one of the parent blocks. The energy of writing variables in \( R_i \) back to memory when \( R_i \) is assigned to \( v \) is already included in the objective function in the second item.

2.2) If \( v \) is not written in \( B \), for the static energy model,

\[ c(n_r(i), n_s(v)) = P(v, R, \sum_{j=1}^{m} B^j)(e_r^R + e_w^M) - P(v, M, \sum_{j=1}^{m} B^j)(e_r^R + e_w^M) \\
-(P(v, R, \sum_{j=1}^{m} B^j) - P(v, R_i, \sum_{j=1}^{m} B^j))(e_r^R + e_w^M) \]  

(24)

where the first term is the energy consumption if \( v \) is assigned to a memory location, the second is the energy of assigning \( v \) to \( R_i \) in \( B \) if \( v \) is in memory in \( B \)'s parent blocks, and the last term is the energy of assigning \( v \) to \( R_i \) in \( B \) if \( v \) is already in a register other than \( R_i \) in \( B \)'s parent blocks.

For the activity based energy model,

\[ c(n_r(i), n_s(v)) = P(v, R, \sum_{j=1}^{m} B^j) e_w^M - P(v, M, \sum_{j=1}^{m} B^j) e_r^M \\
-(1 - P(v, R, \sum_{j=1}^{m} B^j))(\sum_{j=1}^{m} \sum_{u \in V_{Rb^j}|P(u, R_i, B^j) \neq 0} P(u, R_i, B^j)\overline{\Pi}(u, v)C_{rw}^R) \]  

(25)
where the last term is the energy consumed by assigning \( v \) to \( R_i \) which is determined by the hamming distance of \( v \) and the variables that occupies \( R_i \) in any of \( B \)'s parent blocks.

3) For an arc from source node \( S \) to the node of type \( n_r(i) \), i.e., \( a(S, n_r(i)) \), the cost function \( c(S, n_r(i)) \) is simply zero.

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

4.1) If \( v \) is read in \( B \), for static energy model,

\[
c(n_f(v), F) = e_r^M - e_r^R \tag{26}
\]

while for the activity based energy model,

\[
c(n_f(v), F) = e_r^M \tag{27}
\]

4.2) If \( v \) is not read in \( B \), the cost \( c(n_f(v), F) \) is simply assigned to zero in both the static and activity based energy model.

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 \( c(p, q) \) definitions, the objective function for the network flow problem is uniquely defined. The constraints for the network flow problem are defined based on the number of registers available to different types of arcs. They are summarized in the following:

\[
\sum_{i=1}^{i=k} x(S, n_r(i)) = \sum_{v \in (C_y \cup D_y)} x(n_f(v), F) \leq k \tag{28}
\]

\[
\sum_p x(p, z) = \sum_q x(z, q) \quad \forall p, z, q \in N \tag{29}
\]

\[
0 \leq x(p, q) \leq 1 \quad \forall a(p, q) \in A \tag{30}
\]

Applying a network flow algorithm [21] 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 whose lifetimes overlap with the lifetime of \( B \). If the resulted \( x \) value of the 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.

5 Register Allocation for Loop

In this section, we extend our approach to handle programs with loops. A loop can be considered as a control structure with more than one parent blocks merging into it. However, it presents some unique challenges since one of its parent blocks is itself.
One way to solve the allocation problem for loops is to modify the flow problem formulation presented in the last section. Consider a loop with one basic block as shown in Figure 5, where the dash lines represent those variables that are referenced in more than one iteration of the loop execution. We refer to these variables as loop variables and the rest as non-loop variables. The graph for modeling the allocation problem of a loop block can be built following the same guidelines used for the merge case. The graph for the loop block in Figure 5 is shown in Figure 6. (Note that the existence of loop variables does not change the graph since only register nodes are needed to model parent block assignments.) Similar to the merge case, we associate certain probabilities to the variables at the boundary of the loop block and its parent blocks to capture potentially different assignments dictated by different parent blocks. For the example in Figure 5, assuming that the loop has one parent block \(B^{p1}\), we associate \(P(e, R_i, B^{p1})\) and \(P(e, R_i, B^{p2})\) (resp., \(P(f, R_i, B^{p1})\) and \(P(f, R_i, B^{p2})\)) with variable \(e\) (resp., \(f\)), where \(B^{p2}\) corresponds to the loop itself. It may seem that one can now simply use the flow problem formulation for the merge case to find the allocation for a loop. However, the difficulty is that the probabilities associated with each loop variable (e.g., \(P(e, R_i, B^{p2})\)) are dependent on the allocation result of the loop block which is yet to be determined.

![Figure 5: A loop with one basic block, where \(e\) and \(f\) are loop variables, and \(a\), \(b\), \(c\) and \(d\) are non-loop variables.](image)

The optimization problem can still be solved but it is no longer a simple flow problem. In general, assume that \(B^{p2}\) is the loop body itself and that the probability of this block loops back to itself is \(P(B^{p2})\). \(P(B^{p2})\) can be obtained from profiling information.) The probability associated with a loop variable \(v\) can be expressed as follows. If \(v\) is assigned to register \(R_i\) upon exiting the loop, i.e., \(x(n_f(v), F) = 1\), then \(P(v, R, B^{p2}) = P(v, R_i, B^{p2}) = P(B^{p2})\), \(P(v, R_j, B^{p2}) = 0\) for \(j \neq i\), and \(P(v, M, B^{p2}) = 0\). If \(v\) is assigned to memory upon exiting the loop, i.e., \(x(n_f(v), F) = 0\), then \(P(v, R, B^{p2}) = 0\), and \(P(v, M, B^{p2}) = P(B^{p2})\). Consequently, the following expression can be derived:

\[
P(v, R, B^{p2}) = x(n_f(v), F)P(B^{p2})
\]
\[
P(v, M, B^{p2}) = (1 - x(n_f(v), F))P(B^{p2})
\]

It follows that the flow problem in (19) can be transformed to an integer quadratic programming
Figure 6: Flow graph for the loop in Figure 5.

problem. Integer quadratic programming problems are generally NP-hard and may be solved by the branch-and-bound method [26].

We propose an efficient heuristic approach, which generally produces superior allocation results. Recall that our register allocation algorithm determines the variable allocation of a block based on the allocation results of its parent blocks. Consider a loop variable, it may have some initial assignments dictated by the allocation of its parent blocks (other than the loop block itself), another assignment before its first reference in the loop block, and yet another assignment upon leaving the block. To ease our explanation, we refer to these assignments as pre-assignment, in-assignment, and post-assignment, respectively. In Figure 7(a), we show a possible assignment result for the block in Figure 5, where for variable $e$, its pre-assignment is a register, its in-assignment is memory, and its post-assignment is a register. The basis for our approach is the following observations. In general, a loop body is executed frequently, i.e., $P(B^{pl})$ is relatively close to one. If a loop variable’s pre-assignment is different from its in-assignment, only a single switching from memory to register or vise versa is required. However, 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.

Our algorithm employs an iterative approach. With the first allocation attempt, we ignore the fact that the loop body is one of its parents and compute the allocation by simply using the results from the other parent blocks. That is, the value of $P(v, R, B^{pl})$ and $P(v, M, B^{pl})$ in the flow formulation in (19) are assumed to be zero if $B^{pl}$ is the loop body itself and $v$ is a loop variable. We refer to as the initialization phase ($\phi_0$).

The algorithm proceeds as follows. 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 ($\phi_1$) and the third phase ($\phi_2$). In the second phase, we let the in-assignment of each loop variable to be the value obtained from $\phi_0$ and solve the resultant allocation problem. Since the values of $P(v, R, B^R)$ and $P(v, M, B^R)$ are unknown if $v$ is a loop variable, it may seem that we still need to solve an integer quadratic programming problem. However, applying some simple manipulations, we can avoid this difficulty. In particular, we modify the value of cost $c(n_f(v), F)$ for each loop variable $v$. Instead of always setting it to zero as in the merge case, it is defined according to the in-assignment of the same loop variable as follows:

$$c(n_f(v), F) = \begin{cases} 
0 & \text{if } v\text{'s in-assignment is register} \\
-e^M - e^R & \text{otherwise}
\end{cases}$$

The value of $P(v, R, B^R)$ and $P(v, M, B^R)$ for each loop variable $v$ are still assumed to be zero. The modified flow problem correctly captures the overall energy consumption when the in-assignment of each loop variable is set to that obtained from $\phi_0$. The solution to the modified flow problem leads to an improved allocation result.

In the third phase ($\phi_2$), we fix the post-assignment of each loop variable to the value obtained in $\phi_0$. Then, $P(v, R, B^R_i)$ and $P(v, M, B^R_i)$ can be computed by

$$\begin{align*}
P(v, R_i, B^R_i) &= P(v, R_i, B^R) = P(B^R_i) \\
P(v, R_i, B^R_i) &= 0 \\
P(v, M, B^R_i) &= 0 \\
P(v, R, B^R_i) &= 0 \\
P(v, M, B^R_i) &= P(B^R_i)
\end{align*}$$

if $v$'s post-assignment is register $R_i$, if $v$'s post-assignment is memory.

By substituting in (19) the above probability values, we again reduce the allocation problem to a min-cost flow problem. Solving this flow problem will result in an allocation that improves the result obtained from $\phi_0$. The better one from $\phi_1$ and $\phi_2$ 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 $\phi_0$ and following the same procedure as used in $\phi_1$ and $\phi_2$.

In the following, we show that the optimal allocation for a loop block is obtained if our algorithm exits after the initialization phase $\phi_0$. Furthermore, we show that if the second and third phase are needed, their results always improve that from $\phi_0$.

**Lemma 1** Let $E$ be the energy consumption by loop block $B^R_i$ and $E'$ be the energy consumption by $B^R_i$ assuming that the loop variables do not loop back. Then, $\min E \geq \min E'$.

**Proof:** $E'$ can be computed by setting in (19) $P(v, R_i, B^R)$ (for all $1 \leq i \leq k$) and $P(v, M, B^R)$ to zero for each loop variable $v$. Then, $E$ can be expressed as

$$E = E' + \sum_{v \in V_L} \left\{ P(v, R, B^R_i)(e^M_w + e^R_r) - (P(v, R, B^R_i)(e^R_r + e^M_w) - P(v, M, B^R_i)(e^R_r + e^M_w) - ((P(v, R, B^R_i) - P(v, R_i, B^R))(e^R_r + e^M_w))x(n(R_i), n_s(v)) \right\}$$

(31)
where $V_L$ is the set of loop variables. It is not difficult to verify that $E \geq E'$ for any combination of $x(n(R_i), n_s(v))$ values. Hence, $\min E \geq \min E'$.

Based on Lemma 1, we can prove the following theorem.

**Theorem 1** If the allocation result from the initialization phase ($\phi_0$) makes the in-assignment of each loop variable to be the same as its post-assignment, then $\min E = \min E'$.

**Proof:** Consider a loop variable $v$ in $B^{p_i}$. If its in-assignment and post-assignment are both memory, we have $x(n(R_i), n_s(v)) = 0$ and $P(v, R, B^{p_i}) = 0$. Hence, $v$’s contribution to $E$ in (31) reduces to zero. Similarly, if $v$’s in-assignment and post-assignment are both register $R_i$, we have $P(v, R, B^{p_i}) = P(v, R_i, B^{p_i}) = P(B^{p_i})$, $P(v, M, B^{p_i}) = 0$, and $x(n(R_i), n_s(v)) = 1$. Again, $v$’s contribution to $E$ in (31) becomes zero. When more than one loop variable exists, the same conclusion can be obtained. Therefore, if the in-assignment of each loop variable is the same as its post-assignment, we have $E|_{\phi_0} = E'$, where $E|_{\phi_0}$ is the total energy consumption of $B^{p_i}$ using the same allocation as that obtained from $\phi_0$. Since $\min E \geq \min E'$ by Lemma 1, we obtain $\min\{E|_{\phi_0}\} = \min E = \min E'$.

According to Theorem 1, if the in-assignment of each loop variable is the same as its post-assignment, the allocation result from $\phi_0$ is the optimal one for the loop block, and no more allocation phases are needed.

When the in-assignment of some loop variable differs from its post-assignment, the result from $\phi_0$ is no longer optimal. The reason is that the problem formulation used in $\phi_0$ fails to account for the extra energy incurred by the loop variable switching its assignment at the loop boundary. Our second and third phases aim at reducing such energy. In the second phase $\phi_1$, the in-assignment of each loop variable is assumed to be the same as that obtained from $\phi_0$, and the best allocation based on this assumption is identified. Let $E|_{\phi_0}$ be the energy consumption of $B^{p_i}$ when the allocation result from $\phi_0$ is used, and $E|_{\phi_1}$ be the energy consumption of $B^{p_i}$ when only the in-assignments of loop variables obtained from $\phi_0$ are used. After $\phi_1$, we obtain an allocation result having $\min\{E|_{\phi_1}\}$ as the energy consumption. Note that $E|_{\phi_0}$ is simply one of the possible values of $E|_{\phi_1}$. Hence, $\min\{E|_{\phi_1}\} \leq E|_{\phi_0}$. That is, $\phi_1$ always improves the result from $\phi_0$. In the third phase $\phi_2$, we fix the post-assignments of loop variables to the values obtained from $\phi_0$ and find the corresponding best allocation. Similar to the statement made for $\phi_1$, the allocation result from $\phi_2$ also improves that of $\phi_0$.

If more allocation phases are carried out based on the better allocation from $\phi_1$ and $\phi_2$, further improvement may be obtained. 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.

We use the example in Figure 5 to illustrate how our algorithm works for a single-block loop.
Figure 7(a) depicts the allocation result after the initialization phase. Since the in-assignment and post-assignment for loop variables $e$ and $f$ are different, the second and third phase are needed. In the second phase, cost $c(n_{f2}(e), F)$ of the flow graph in Figure 6 is set to $-c_{w}^{M} - c_{r}^{R}$ as the in-assignment of $e$ is memory, while cost $c(n_{f2}(f), F)$ is set to zero as the in-assignment of $f$ is register. The allocation result after the second phase is shown in Figure 7(b). In the third phase, we let $P(e, R_{1}, B^{p}) = P(B^{p})$, $P(e, R_{2}, B^{p}) = P(e, M, B^{p}) = 0$ since the post-assignment of $e$ is register $R_{1}$, and $P(f, R, B^{p}) = 0$, $P(f, M, B^{p}) = 0$ since the post-assignment of $f$ is memory. The allocation result after the third phase is the same as that after $\phi_{1}$. Since the in-assignment of $e$ (resp., $f$) is the same as its post-assignment in the allocation result, our process stops. It is obvious that the allocation result is optimal for this block.

Our algorithm can be readily extended to handle the case when there are more than one block inside a loop body, i.e., the control structure inside a loop body contains branches and loops. For branches and merges, each phase in our algorithm needs to solve the allocation problem for each block in the loop body following the process discussed in Section 3 and 4. In $\phi_{1}$ (resp., $\phi_{2}$), the in-assignments (resp., post-assignments) of loop variables obtained from $\phi_{0}$ are used to compute the necessary cost or probability values. For loops inside a loop, the sequence of the allocation is determined by the nested levels of loops. Since inner loops are executed more frequently than outer loops, the inner-most loop is allocated first and the result is used by the outer loops.

![Figure 7: The allocation results of different allocation phases for a loop. The solid lines correspond to variables in registers, while the dash lines correspond to variables in memory.](image)

6 Experimental Results and Discussions

Graph-coloring based techniques such as the ones in [6] are most often used for solving the register allocation problem. We compare our algorithm with a graph-coloring approach. We have implemented a graph-coloring register allocator and our block-by-block register allocator in the Tiger compiler presented in [1]. The modularity of the Tiger compiler makes it easy to fix other phases of the compiler, while only alternate the register allocation algorithms. The procedures for finding the basic blocks, building the trace, and computing the lifetime of each variable are kept...
the same in both allocators. In the implementation of the graph-coloring allocator, an interference graph is built for each program. In applying the graph-coloring approach to color the graph, if there is no node with degree less than the number of available registers in the graph, the node with the highest degree will be spilled until all the nodes are colored or spilled. In the implementation of our algorithm, a network flow graph is built for each block. Then our algorithm for different control structures is applied to each basic block in the topological order to find the best allocation.

We compare the allocation results of the two different approaches for several real programs containing complex control structures. The numbers of memory references by the two allocators are summarized in Table 3. The improvement in the energy consumption and the number of memory references by using our algorithm over the graph-coloring approach is summarized in Table 4. According to data in [3, 17, 12], it is reasonable to assume that the ratio of the average energy for memory reference over the average energy for register reference, $e_M/e_R$, is in the range of 10 to 30. In Table 5, we summarize the sizes of allocation problems that the two approaches need to solve. As stated at the beginning of the paper, the complexity of our algorithm is $O(b \cdot \text{time complexity})$, where $b$ is the total number of blocks and $l(n, k)$ is the time complexity for solving the register allocation problem for a single block that has $n$ variables and $k$ available registers. The function $l(n, k)$ is either $O(n \log n)$ for the single-read static-energy model, $O(kn \log n)$ for the multiple-read static-energy model, or $O(kn^2)$ for the activity based energy model [5, 13].

Table 3: The number of memory references allocated by two approaches.

<table>
<thead>
<tr>
<th>Example Programs</th>
<th>k</th>
<th># of register candidates</th>
<th># of data references</th>
<th>Our approach # of spilled candidates</th>
<th># of memory references</th>
<th>Graph-coloring # of spilled candidates</th>
<th># of memory references</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>16</td>
<td>106</td>
<td>1218</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>284</td>
</tr>
<tr>
<td>factorial</td>
<td>14</td>
<td>36</td>
<td>309</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>92</td>
</tr>
<tr>
<td>branch</td>
<td>14</td>
<td>28</td>
<td>67</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 4: The improvement of results by our algorithm over graph-coloring

<table>
<thead>
<tr>
<th>Programs</th>
<th>$e_M/e_R = 10$</th>
<th>$e_M/e_R = 15$</th>
<th>$e_M/e_R = 30$</th>
<th>Improvement of # of memory references (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>67.8</td>
<td>76.4</td>
<td>87.1</td>
<td>23.3</td>
</tr>
<tr>
<td>factorial</td>
<td>72.8</td>
<td>80.7</td>
<td>89.6</td>
<td>29.8</td>
</tr>
<tr>
<td>branch</td>
<td>68.2</td>
<td>77.0</td>
<td>87.4</td>
<td>23.9</td>
</tr>
</tbody>
</table>

Table 5: The size of problems solved by two approaches.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Interference graph # of nodes</th>
<th># of edges</th>
<th>Network flow # of blocks $b$</th>
<th>max # of variables $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>106</td>
<td>1296</td>
<td>16</td>
<td>48</td>
</tr>
<tr>
<td>factorial</td>
<td>36</td>
<td>385</td>
<td>5</td>
<td>27</td>
</tr>
<tr>
<td>branch</td>
<td>28</td>
<td>293</td>
<td>4</td>
<td>26</td>
</tr>
</tbody>
</table>
The experiment results show that our algorithm can achieve more than 60% improvement in the energy consumption due to data references of a program over a graph-coloring approach. At the same time we also saved more than 20% of the memory references which will save the execution time due to data references of a program. Furthermore, our approach is simple to use and the running time is polynomial.

There are some other improved graph-coloring based algorithms, such as the one in [11]. The register allocation results by those improved algorithms are better than the simple graph-coloring approach implemented here. But again, those algorithms are all proposed to optimize the execution time, not the energy consumption, of a program.

Our algorithm proposed in this paper does not take into account any look ahead information while doing register allocation. It also does not allow any backtracking. In some cases, our approach may produce suboptimal allocation results. One simple extension of the algorithm would be to change the cost \(c(n_J(v), F)\) associated with the arc \(a(n_J(v), F)\) for a cross boundary variable \(v\) according to the possible allocation results of \(v\) in child blocks.

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 experimental results by using our algorithm so far are very promising. More experiments with larger code sizes are being conducted. We are also investigating on how to deal with procedure calls.

7 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.

8 References


[27] T.A. Proebsting and C.N. Fischer, “Demand-driven register allocation,” ACM Transactions on 
Programming Languages and Systems, vol. 18, no. 6, November 1996, pp. 683-710.
126-128.
[29] N. Woo, “A global dynamic register allocation and binding,” Design Automation Conference, 
1990, pp. 505-510.