Binary Translation to Improve Energy Efficiency through Post-pass Register Re-allocation

Kun Zhang Tao Zhang Santosh Pande
College of Computing
Georgia Institute of Technology
Atlanta, GA, 30332
{kunzhang, zhangtao, santosh}@cc.gatech.edu

ABSTRACT
Energy efficiency is rapidly becoming a first class optimization parameter for modern systems. Caches are critical to the overall performance and thus, modern processors (both high and low-end) tend to deploy a cache with large size and high degree of associativity. Due a large size cache power takes up a significant percentage of total system power. One important way to reduce cache power consumption is to reduce the dynamic activities in the cache by reducing the dynamic load-store counts. In this work, we focus on programs that are only available as binaries which need to be improved for energy efficiency. For adapting these programs for energy-constrained devices, we propose a feed-back directed post-pass solution that tries to do register re-allocation to reduce dynamic load/store counts and to improve energy-efficiency. Our approach is based on zero knowledge of original code generator or compiler and performs a post-pass register allocation to get a more power-efficient binary. We attempt to find out the dead as well as unused registers in the binary and then re-allocate them on hot paths to reduce dynamic load/store counts. It is shown that the static code size increase due to our framework is very minimal. Our experiments on SPEC2000 and MediaBench show that our technique is effective. We have seen dynamic spill loads/stores reduction in the data-cache ranging from 0% to 26.4%. Overall, our approach improves the energy-delay product of the program.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors - *Compilers, optimization*

General Terms
Performance, Measurement, Design, Experimentation

Keywords
Register re-allocation, dead registers, unused registers, cache power consumption

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

1. INTRODUCTION
Energy efficiency has become a first class optimization parameter for both high performance processors as well as embedded processors; however, a very large percentage of legacy binaries exist that are optimized for performance without any consideration to power. In this work, we focus on improving such binaries for popular processors such as the Pentium (x86) for saving power without degradation (in fact slight improvement) in performance.

Because cache size and organization are so critical to the overall performance of a processor, modern processors tend to deploy caches with large sizes and high degree of associativity. A large cache consumes a significant part of the total processor power. As an example, the high-associativity low-power caches used in ARM processor family take almost half of the power budget of the processor [16]. There has been research on innovative cache design to reduce the power consumption of a cache [9, 10, 11]. In our work, we try to reduce cache power consumption from another point of view. Our approach reduces dynamic activities of a cache by reducing the number of dynamic loads/stores instructions that tend to hit L1 D-cache. We attempt such an optimization using feedback directed approach that performs register re-allocation.

Register allocation is the process of deciding how the variables inside a program are assigned to the physical registers of the machine. Generally, there are two main approaches for performing register allocation [3]. The first one assumes that variables reside in memory and they are placed in registers when they are allocated. Under this model, the register allocator does not spill a value. The second model assumes that all the variables are represented as symbolic registers. Any symbolic register that cannot be mapped to a physical register has to be spilled. The task of a register allocator is to map the infinite symbolic registers to finite physical registers such that the lifetime of symbolic registers bound to the same physical register will not overlap. We base our approach on the second register allocation model since this paradigm is popular. Based on the method they employ, register allocators can be roughly divided into two categories: based on graph-coloring algorithm [1, 2] or based on live range splitting [3]. Hybrid register allocators also exist [4]. In a graph-coloring register allocator, first an interference graph is constructed in which the nodes represent symbolic registers. If two symbolic registers are simultaneously live then an edge is added between them. Graph-coloring based register allocators attempt to color the interference graph with a limited number of colors, less than or equal to the number of physical registers in the machine. Graph-

coloring is a NP-hard problem, thus, the allocators have to apply heuristics to achieve a (sub-optimal) solution. In case there are variables that cannot be colored, they have to be spilled. Spilling a variable means storing the value of the variable into the memory rather than register. In splitting based allocators, when a live range cannot be colored, it attempts to split it so that parts of it can be colored and other parts are spilled. Spill load/stores lead to memory accesses that sometimes are a significant portion of total dynamic load/stores. Thus, if we reduce the dynamic spill load/stores significantly, we can reduce the dynamic activities and the power consumption of the data cache where these load/stores mostly hit. Cycle counts also improve (albeit to much lesser extent especially for out of order processors) and overall, energy-delay product improves.

Opportunities for post-pass optimizations. Due to relative maturity of register allocators, the natural question to ask is: do opportunities exist for post-pass optimizations? The answer is yes mainly due to the following reasons. In spite of a lot of research in the register allocation area, both graph-coloring and splitting based allocators *leave a number of dead and/or unused registers in places where spills exist in the generated code.* . Secondly, almost no allocator tends to factor in power issues during allocation; such issues involve moderating code growth but still achieving spill reduction on hot paths. In this paper, we propose register re-allocation approach based on the above observations to tackle the limitations of traditional allocators. There are two potential approaches to solve the problem: first one involves undertaking the whole program register re-allocation that analyzes the tradeoff between hot and cold paths and removes as many spills as possible on hot paths at the expense of cold paths; the second one is an incremental approach which does not disturb the original allocation but performs necessary *fixes* to the allocation on hot paths leaving original allocation mostly undisturbed on cold paths. We rule in favor of second approach here mainly due to the following observations:

- Almost 70% of the typical code is cold and 30% is hot; thus, an approach which tends to favor spills on cold parts to remove the ones on hot parts would result in a large code growth. Code growth is not usually desirable for most embedded devices and moreover excessive code growth could adversely impact I-cache performance and degrade the power consumption. Our preliminary implementation validated this hypothesis.

- Performing the whole program re-allocation is expensive and is also difficult as a post-pass approach.

- Incremental solution such as ours is fast and does not need a lot of analysis and could be ideally adopted as a post-pass due to such attributes.

Scope of our work. Motivated by above observations and the fact that there are many programs only available as binaries that need to be improved for energy efficiency we propose an incremental post-pass solution which is based on zero knowledge of the compiler or code generator originally used to generate the binary. Our scheme discovers program behavior and then adapts the binary by incrementally changing register allocation where opportunities exist. This problem is quite important for popular processors such as Pentium that have only a few registers. Also due to the relative ease of binary portability, x86 compatible embedded processors are gaining popularity now such as the AMD Elan SC410, Cyrix MediaGX, and National Semiconductor Geode etc., making this problem important.

The remainder of the paper is organized as follows: section 2 elaborates on the limitations of the traditional register allocators motivating a need for post-pass solution; section 3 gives an overview of our framework; sections 4 presents the details of our approach; section 5 shows the results of our exoeriments; section 6 discusses the related work; finally section 7 concludes our paper.

2. BACKGROUND AND MOTIVATION

The preceding section discussed pros and cons of different possible approaches. We now get into details of possible opportunities available to the post-pass incremental approach when the original register allocation is based on coloring and/or live range splitting.

The early coloring allocaters do not undertake splitting scheme, so they spill live ranges completely. Such approaches leave a large number of dead and/or unused registers. This phenomenon occurs because a register may be free between two disjoint live ranges which are assigned to it. However, even a splitting-based register allocator could leave dead/unused registers sometimes. We will show two examples to explain why our technique will do optimization in the presence of either one.

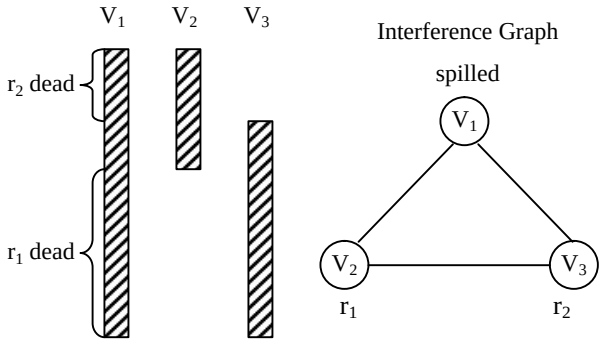

Figure 1. Register allocation without splitting

Figure 1 illustrates the dead register problem in a register allocator without a splitting mechanism. In the example live ranges V_1, V_2, V_3 interfere with each other as shown through the interference graph. Suppose we have only two registers. One of the live ranges must be spilled. Assume that V_1 has the least spill cost and gets spilled; register r_1 is allocated to variable V_2 and register r_2 is allocated to V_3. Figure 1 shows that r_2 and r_1 are dead for certain parts of live range V_1. Specifically r_2 is dead before it gets the value V_3 and r_1 is dead after the last use of V_2. If there are spills for V_1 in these sub-ranges, we can simply re-allocate r_1 or r_2 to avoid the spills. Note a splitting-based register allocator will do the similar work and eliminate the problem shown in Figure 1.

In the second example shown in Figure 2, there are five live ranges V_1, V_2, V_3, V_4 and V_5. Assume that we are using a splitting type register allocator based on Cooper&Simpson's paper [5] and suppose we have only two registers r_1 and r_2. According to this splitting scheme, nodes in the containment graph represent live ranges and an edge from V_j to V_i in the graph indicates that V_i is

live at a definition or use of V_j. In this example, the nodes are V_1, V_2, V_3, V_4 and V_5, and the containment graph is shown in Figure 3. V_1 and V_2 are in a loop and have higher spill costs than V_3. So they will be allocated to registers r_1 and r_2. There is no edge <V_3, V_1> in the containment graph and the splitting cost is less than the spilling cost for V_3. So V_3 will be split around live range V_1 as shown in Figure 2 and r_1 will be allocated to V_3'. Note that r_1 is carrying a dead value in B_2, so it can be used for re-allocation. There are three live ranges in B_2 and V_3 is in r_1, so one of V_4 and V_5 has to be spilled. In that case, we can allocate r_1 to the spilled live range and remove the corresponding load in hot basic block (B_2) while inserting a load in cold basic block (B_1). Now consider another important splitting based allocator proposed in Chow&Hennessy's paper [3]. Due to its forbidden list becoming full, the priority based coloring scheme will split live range V_3 as shown into V_3' (please refer to Figure 2). Although V_3' has no use in B_2, it will get a register in B_2 and in turn could spill values in B_2 that appear later in priority list.

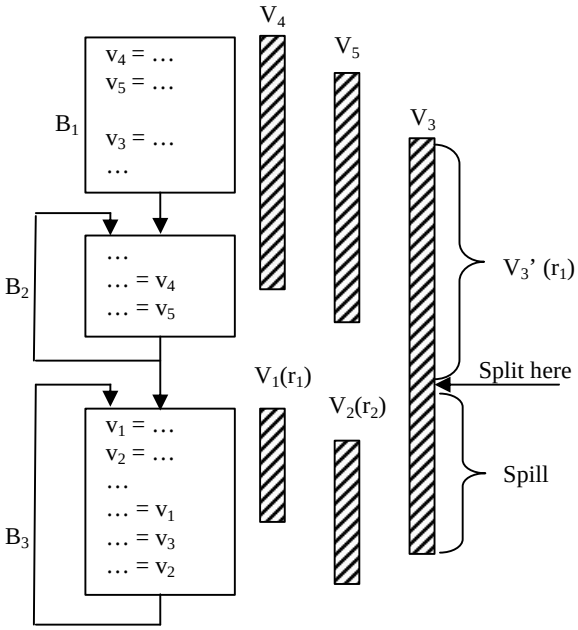

Figure 2. Register allocation with splitting mechanism

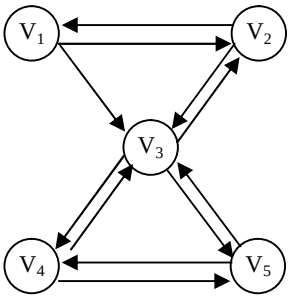

Figure 3. Containment graph for the example in Figure 2

Sparse or long live ranges could also lead to inefficiencies in register allocators. A variable may be used so many times along the live range that it gets a physical register. However, the variable is not necessarily used evenly throughout the live range, possibly leaving some large regions where the variable occupies a register but is not used. We call such registers as unused registers in these regions. The unused register problem is common under almost all register allocators. In this work, we perform analysis to discover the availability and profitability of dead/unused registers in program binaries to improve the energy efficiency.

3. FRAMEWORK OVERVIEW

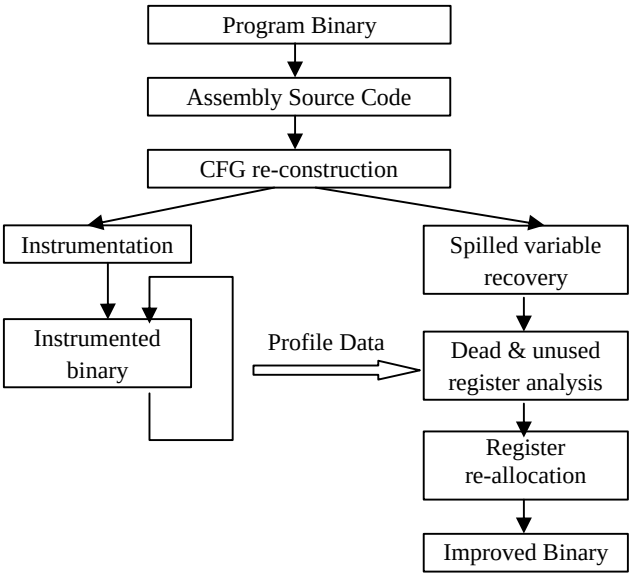

Figure 4. Our approach

Our framework for post-pass register re-allocation is shown in Figure 4. Starting from the program binary with zero knowledge of the compiler generating it, we first dis-assemble the binary to get assembly source code. Then we re-construct the program control flow graph (CFG) from the assembly code. We instrument the binary based on the CFG to get the basic block execution frequency counts. The instrumented binary is executed with many different inputs (called training runs) to collect the program profile data. The profile data is fed back to the register re-allocation pass to assist the cost and benefit analysis of dead/unused register re-allocation.

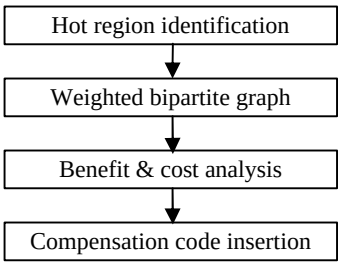

Figure 5. Register re-allocation overview

The details for register re-allocation pass are further illustrated in Figure 5. First hot regions in a program are identified which our optimization focuses on. Then the re-allocation of registers inside hot regions is modeled as a weighted bipartite graph matching problem. The benefit of re-allocating a register and the cost of

corresponding compensation code (loads/stores inserted into cold blocks) are analyzed and the re-allocation with maximum overall benefit is chosen. Finally, the compensation code is inserted to guarantee the correctness of the program.

4. REGISTER RE-ALLOCATION

In this section, we will elaborate our register re-allocation framework introduced in last section. Our re-allocation framework includes both dead register (a register which does not contain a live value) re-allocation and unused register (a register which may contain a live value but is neither defined nor used at current basic block) re-allocation.

4.1 Dead and Unused Registers

First, we will introduce some definitions about dead and unused registers and show how to identify dead registers through data flow analysis.

Definition1: Unused register in a basic block is a register which is neither defined nor used within that basic block.

Define Universal_set be the set of all available registers for allocation; used[B] be the set of registers which are defined or used in B; unused[B] be the set of unused registers in B. It is obvious that, unused[B] = Universal_set – used[B]. Please note that an unused register in block B may or may not be carrying a live value in it.

Definition2: A register is dead at a program point p if it does not carry a live value at p.

For a basic block B, denote dead_in[B] to be the set of dead registers at the entry of B; dead_out[B] to be the set of dead registers at the exit of B. The algorithm to collect dead register information is shown in Figure 6.

Initialization:

dead_in[B] = dead_out[B] = Universal_set;
dead_gen[B] = {r | r is defined before used in B};
dead_kill[B] = {r | r is used before defined in B};

Data flow equations:

$dead\_out[B] = \bigcap_{S \text{ is successor of } B} dead\_in[S]$;

$dead\_in[B] = dead\_gen[B] \bigcup (dead\_out[B] - dead\_kill[B])$;

Figure 6. Dead register analysis

It is similar to live variable analysis [17] except that in computing dead_out[B], the unification operator is "$\bigcap$ " instead of "$\bigcup$ " since a register is dead out of a block B only if it is dead along every path from the exit of B. Also note that even if a register is dead at the beginning of a basic block, it may not be used for register re-allocation directly. Since it may be defined and used again inside the basic block. So we consider the set (dead_in[B] – used[B]) for dead register re-allocation instead. Note that even if a register is dead at the beginning of a basic block, it may not be used for register re-allocation directly since it may be defined and used again the basic block. So we consider the set (dead_in[B] – used[B]) for dead register re-allocation in blokck B instead. Please note that a dead register need not be stored before it is loaded but an unused register (if carrying a live value) must be stored before a new value is loaded in it. Finally, this value must be restored in it at the right program point (before its use). These issues form the basis for compensation code and determine profitability for the use of dead/unused registers.

4.2 Hot region identification

Next, in this phase, we identify hot regions in a program. The goal of the algorithm is to reduce the dynamic spills inside a hot region maybe at the cost of increased dynamic spills in cold basic blocks. Since hot regions tend to be executed more frequently than cold blocks, we will achieve overall performance and energy efficiency improvement.

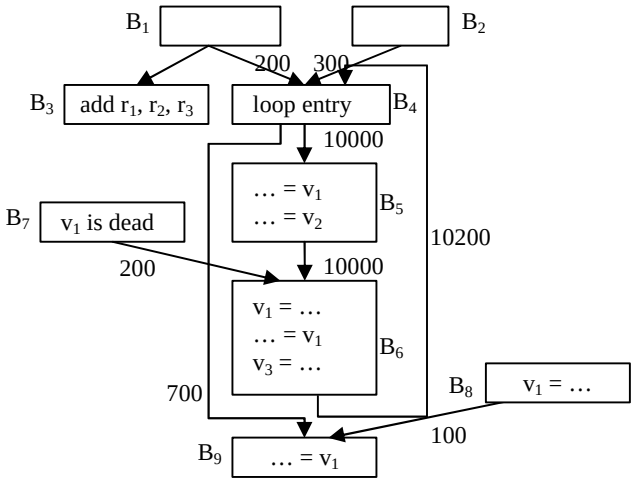

Figure 7. An example

To identify hot regions, first we need to identify hot basic blocks. Currently, hot basic blocks are identified in a simple way in our framework. We consider those basic blocks whose average dynamic execution frequency exceeds a certain threshold as hot basic blocks. The threshold is the average execution frequency of the basic blocks. Next, adjacent hot basic blocks are merged to form hot regions. An important property of our algorithm is that it forms the regions in such a way that any two hot regions must be separated by cold basic blocks. In other words, for disjoint hot regions register re-allocation can be carried out independently. As an example, consider the sub-CFG shown in Figure 7. In the example, basic blocks B_4, B_5 and B_6 belong to a loop body and they are hot basic blocks. The threshold is 8000. According to the hot region identification algorithm, B_4, B_5 and B_6 form a hot region. The other basic blocks shown are cold. The detailed algorithm for hot region identification is shown in Appendix.

4.3 Spill identification

Next we identify the spills to be removed from hot blocks. Please note that not all the loads/stores constitute spills. It is not safe to keep pointer variables in registers in the absence of aggressive alias analysis and thus, such values are loaded and stored from memory. In our framework, we perform a simple alias analysis based on the one proposed in [17] to determine the singly aliased variables (this is a conservative analysis which assumes all pointer variables are multiply aliased unless proved otherwise through reaching definition analysis). Only the loads/stores corresponding

to singly aliased, non-array variables are considered as spills and are used by subsequent passes. This assumption is safe and also quite accurate since very rarely compilers perform register allocation for array variables.

4.4 Weighted bipartite graph matching

In this section, we explain how to model the register re-allocation inside a hot region as a weighted bipartite graph matching problem.

For a hot region, a bipartite graph is formed in the following way. The spilled variables in a hot region are gathered to form one bipartition of the bipartite graph. The same variable in different basic blocks is represented as different vertices. This separation is deliberately done to give a finer control over allocating two instances of the same spill value independently. The other bipartition is formed by the set of all dead registers in the hot region. Again, the same register in different blocks is represented through different vertices. If a variable and a register are in the same basic block and they are of the same type (in terms of bit width), then we add an edge in the bipartite graph to connect them. That means that the variable could be possibly allocated to the register. Each edge has a weight associated with it. The weight of the edge is defined by the dynamic spill loads/stores caused by the corresponding variable in the corresponding basic block. An edge's weight thus indicates the dynamic spill loads/stores that will be eliminated if we re-allocate the variable into a register. Again, consider the sub-CFG in Figure 7. As mentioned before, B_4, B_5 and B_6 form a hot region. Assume v_1 and v_2 are spilled in B_5; v_1 and v_3 are spilled in B_6. There is a dead register r_1 in B_5 and dead registers r_1 and r_2 in B_6. The resulting bipartite graph is shown in Figure 8.

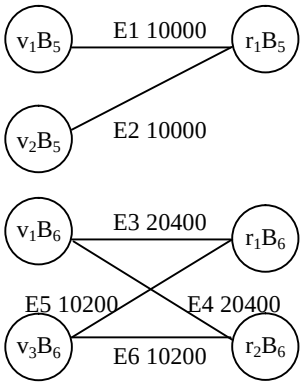

Figure 8. Bipartite Graph

Next we discuss how to perform register re-allocation inside a hot region; the costs and benefits are weighed in to perform matching. Note that we would like to avoid assigning one register to multiple variables in a hot region because doing that will introduce compensation code inside the hot region. Due to the same reason, we also like to avoid assigning one value to multiple registers in a hot region. Combining these two conditions, any viable solution to register re-allocation inside a hot region corresponds to a matching in the bipartite graph. However, the benefit brought by register re-allocation is not free. We often need compensation code to maintain the correctness of the program (we will discuss the computation of compensation cost in detail later) Thus, the overall benefit can be expressed as the benefit of reassignment minus the compensation code cost. To save energy, we need to choose that matching which has maximum overall benefit. Although weighted bipartite graph maximum matching itself is easy, it is not so if we have to consider compensation cost for finding the best solution. We have to compute the compensation cost for each matching to decide the overall best solution. To guarantee optimality, in our current implementation, we take a brute-force approach. We try all the matchings, compute the corresponding compensation cost then find the optimal one. However, due to the small size of hot region and the two conditions mentioned above, this phase does not take too much time as shown in our experimental results later.

4.5 Compensation code cost analysis

To analyze the compensation code cost of register re-allocation, first, we find out the edges in the bipartite graph which are incident on the same variable and the same register. We then check the corresponding basic blocks of those edges. If any of the corresponding basic blocks are adjacent in CFG, we combine the edges to be analyzed together and call the resulting edge set as an analysis unit. The motivation for combining the edges is that, in such a case, it is not necessary to introduce any compensation code between the corresponding basic blocks. As an example, in bipartite graph shown in Figure 8, we can combine E1 and E3 as an analysis unit. If we reassign r_1 to v_1, we do not need any compensation code between B_5 and B_6.

Next, for each analysis unit, three regions are created to assist analysis. The basic blocks corresponding to the edges in an analysis unit form a region named cur_region. Cur_region is a subset of the hot region. It is possible that the cur_region could lead to the insertion of compensation code in hot regions. To avoid this problem, we extend cur_region to include as many hot blocks as possible within the hot region. Thus we have more opportunity to insert loads/stores in cold blocks. A block B in the hot region will be added to cur_region if its predecessors and successors are all in cur_region and the dead register r considered for re-allocation belongs to dead_in[B]. The first condition together with the condition of keeping a spilled variable in one register in a hot region mentioned earlier will assure that expanding cur_region will not impact other analysis units. The second condition is to guarantee that r is always dead in cur_region. Otherwise, r may contain a live variable in cur_region which will lead to extra compensation code costs. We define pred_region/cur_region as a set of blocks which are not in cur_region, and have successors/predecessors in cur_region. As an example again, consider the analysis unit {E1, E3} in bipartite graph in Figure 8. At first cur_region includes basic blocks B_5 and B_6. Then B4 will be added to cur_region through the expansion. So cur_region is made of B_4, B_5 and B_6. Pred_region includes basic block B_1, B_2 and B_7. Succ_region includes basic block B_9. For dead register re-allocation, compensating load instructions can be inserted in pred_region or cur_region. Compensating store instructions can be inserted in succ_region or cur_region. In the description, we will use four variables w_cur_load, w_cur_store, w_pred_load, w_succ_store to represent the cost of inserting compensating load in cur_region, compensating store in cur_region, compensating load in pred_region and compensating store in succ_region respectively. Next we will analyze how to

compute the values of these variables case by case. First, consider the compensating load instruction. To re-allocate a dead register to a spilled variable, we may need to load the spilled variable into the register first. The load instruction could be inserted into the entries of cur_region, or it could be hoisted out of the basic blocks in cur_region.

```
w_cur_load = 0;
for each entry E of cur_region
   // if the variable is dead before entering E, do nothing
   if spilled variable is live at the entry of E
      w_cur_load += average execution times of E;
   EndIf
EndFor
```

Figure 9. W_cur_load computation

Consider the analysis unit {E1, E3} in bipartite graph in Figure 8. It is easy to see that to allocate dead register r_1 to v_1, we could insert a load r_1, v_1 at the entry of B_4. Also, we could hoist such an instruction out of the loop body. We have to be careful here. First, we may not be able to simply insert the load at the end of the predecessor basic blocks in pred_region. Consider the same example. We cannot insert a load r_1, v_1 at the end of B_1 because r_1 is used in B_3 which is one of the successors of B_1. In that case, we have to create a new basic block and insert it between B_1 and B_4. Besides the load instruction cost, we have to count the jump instruction cost too. Currently, we regard jump instruction has the same cost as load/store instruction. Another interesting point is that it is not necessary to insert a load into a predecessor basic block if the spilled variable is dead at the end of the basic block. For example, consider B_7 in pred_region. Since v_1 is actually dead at the end of B_7 so there is no need to do anything. The analysis for inserting compensation load in B_2 is the same. For the cases in which we could either choose from creating a new basic block or inserting a load instruction directly, the lower cost solution would be chosen. The algorithms to compute w_cur_load, i.e., the cost of inserting loads in cur_region and w_pred_load, i.e., the cost of inserting loads out of cur_region are shown in Figure 9 and Figure 10.

The analysis of compensation store instructions is quite similar to load instructions though more complicated. Again consider the example in Figure 7. Suppose we re-allocate dead register r_1 to v_1 in the hot region. We have to store r_1 back to v_1 finally. However, we cannot insert the store instruction in B_9 since v1 is actually live at the entry of B_9 and the value of v_1 may be defined in basic block B_8 at runtime. If we insert the store instruction in B_9, it may wrongly overwrite the correct value of v_1. In that case, we will have to insert a new basic block between B_4 and B_9. In computing compensation cost, an important issue is that in most programs the spilled variables will cause much more loads than stores. This property enables us to gain more benefit. If there is no store instruction in an analysis unit, we do not need compensation store even if the variable is live out of cur_region since the spill variable's value is not changed. We will check the basic blocks corresponding to the edges in an analysis unit to see if there is any store instruction for the spilled variable we want to do re-allocation. If such store instruction exists, the algorithms to compute w_cur_store, i.e., the cost of inserting stores in cur_region and w_succ_store, i.e., the cost of inserting stores out of cur_region shown in Figure 11 and Figure 12 will be used for store compensation cost analysis.

Now we have known the benefit and the compensation cost of a matching, so we can calculate the overall benefit of a particular matching. As mentioned before, we will just take a brute force approach and choose the matching having maximum overall benefit.

```
w_pred_load = 0;
For each basic block P in pred_region
   // if the variable is dead thereafter, do nothing
   If spilled variable considered is live at the exit of P
      new_bb_cost = 0;
      For each successor S of P in cur_region
         // if the variable is dead before entering S,
         // no need to insert load
         If spilled variable considered is live at the entry of S
            // assume the same cost of load and jump
            new_bb_cost += 2 × avg. execution times of edge P→S;
            // have to insert new basic blocks or
            // inserting new basic blocks is actually cheaper
            If (the register to be re-allocated is not in dead_out [P]
                  || new_bb_cost < average execution time of P)
               w_pred_load = w_pred_load + new_bb_cost;
            EndIf
         EndIf
      EndFor
   // insert a load at the end of P
   Else
      w_pred_load = w_pred_load + avg. exec. times of P;
   EndIf
EndFor
```

Figure 10. W_pred_load computation

```
w_cur_store = 0;
for each exit E of cur_region
            // if the variable is dead thereafter, do nothing
            if spilled variable is live at the exit of E
w_cur_store += average execution times of E;
            EndIf
EndFor
```

Figure 11. W_cur_store computation

4.6 Unused Register Allocation

The algorithm for dead register re-allocation is not suitable for unused register re-allocation since the value in the unused register is usually live. Thus, the original value in the unused register has to be stored back to the memory before re-allocation and loaded from memory to the register after re-allocation. This is done to make sure that there is a correct residency of values in the register before respective points of their uses. This increases compensation cost greatly. However, we have another algorithm to save energy.

```
w_succ_store = 0;
For each basic block S in succ_region
  // if the variable is dead before entering S, do nothing
  If spilled variable considered is live at the entry of S
    live_on_cold_pred = false;
    For each predecessor P of S not in cur_region
      If (spilled variable considered is live out of P)
        live_on_cold_pred = true;
      EndIf
    EndFor
    new_bb_cost = 0;
    For each predecessor P of S in cur_region
      // if the variable is dead thereafter, no need to insert store
      If spilled variable considered is live at the exit of P
        // assume the same cost of load and jump
        new_bb_cost += 2 × avg. execution times of edge P→S;
        // have to insert basic blocks or
        // inserting new basic blocks is actually cheaper
        If (live_on_cold_pred
              || new_bb_cost < avg. exec. times of S)
          w_succ_store = w_succ_store + new_bb_cost;
        EndIf
      EndIf
    EndFor
  // insert a store at the entry of S
  Else
    w_succ_store = w_succ_store + avg. exec. times of S;
  EndIf
EndFor
```

Figure 12. W-succ_store computation

4.6.1 Unused register region identification

The region for unused re-allocation will be formed based on both the profiling and unused register information. For a register r which is available for allocation, adjacent basic blocks which all have r as unused register will be merged to form a region, denoted by unused register region. That means in such a region, r is always unused. Among the regions, those which include hot blocks will be considered for re-allocation.

4.6.2 Compensation cost analysis

The benefit of re-allocation in an unused register region is defined similarly as dead register re-allocation. However, the compensation cost analysis is more complicated.

In dead register compensation cost computation, we only need to consider the load instruction which loads variable into the re-allocated register (spill_var_load) and the store instruction which stores the variable back to memory (spill_var_store). Under unused register compensation cost computation besides the previous two cases, we have to consider the store instruction which stores the original value in the unused register back to memory (orig_var_store) and the load instruction which loads the saved value from memory back to the unused register (orig_var_load). In dead register re-allocation, spill_var_load can be inserted in pred_region or cur_region; spill_var_store can be inserted in cur_region or succ_region. In other words, there are only four situations which need to be considered. In unused register allocation, spill_var_load and orig_var_store could be inserted in pred_region or cur_region; spill_var_store and orig_var_load could be inserted in cur_region or succ_region. The constraints are that orig_var_store has to precede spill_var_load and spill_var_store has to precede orig_var_load. the method for analyzing compensation cost is the same as in dead register re-allocation, to save space, we have not explicitly listed it here.

4.6.3 Unused register region ordering

Unlike the hot regions for dead register re-allocation, the unused register regions need not be disjoint. Thus, we use some ordering on processing the regions. The region with maximum benefit is selected first. Then the unused register regions, which have common basic blocks with it, are deleted from the region set. After this step, again the next region with maximum benefit is selected until all the regions are processed.

5. EXPERIMENTS AND RESULTS

Our experiments are based on x86 (Pentium) architecture. It has only six general purpose registers (eax, ecx, edx, ebx, esi, edi). The other two registers esp and ebp are stack pointer register and frame pointer register respectively and are not used for re-allocation. In our experiments, CFGs for the benchmark binaries are re-constructed first. Then we use 5 different input data sets (training run) to gather basic block profile data which is fed back to re-allocation pass. After re-allocation pass is performed, the optimized binaries are evaluated by another input data set (test run) different from the previous 5 input data sets (training runs).

The work was implemented in the Machine Suif compiler [18], a research infrastructure for profile-driven and machine-specific optimizations. Machine Suif can be used to instrument programs for profiling and carries out various code optimizations. It uses the algorithm described by George&Appel [19] to do register allocation. The algorithm interleaves Chaintin-style [1] simplification steps with Briggs-style [2] conservative coalescing and eliminates move instructions while guaranteeing not to introduce spills. Thus, Machine Suif has a very good register allocator which is highly optimized. That is why we used it to generate the base level binaries although it does not support live range splitting.

The benchmarks are chosen from SPEC2000 integer benchmarks and MediaBench [13] embedded benchmarks. The original binaries of the benchmarks are produced by Machine Suif. Then our register re-allocation algorithm is performed. The ref input set is used for generating results for SPEC2000 benchmarks. Here not all the benchmarks are selected because Machine Suif could not compile some of the benchmarks. Crafty, gap etc. can be compiled, but the binaries do not execute; gcc, perlbmk, vortex, mpeg2 (de) etc. fail in the do_gen pass in current implementation of Mach Suif.

Table 1 shows the number of hot/cold basic blocks in each benchmark binary, the number of spill loads/stores originally present inside hot basic blocks and those which are removed by our dead /unused register re-allocation approach. We can see that the number of hot basic blocks is either relatively small or the number of overall basic blocks is not big (such as adpcm encoder and decoder). Thus, the sizes of hot regions are normally small too. This is consistent with the general observation that the

majority of the execution time of a program is spent in a small region of the code. Due to small sizes of hot regions, the brute force approach in our re-allocation algorithm normally does not cause too much run time overhead. The time taken by our post-pass reallocator is shown in Table 2. One can see that except for Twolf and Vpr the time taken for re-allocation is negligible. Twolf and Vpr are both very large benchmarks consisting thousands of basic blocks and so the time required to reallocate is high.

Table 1 also shows the breakdown of spills removed due to dead as well as unused registers. One can see that there is a good amount of spill to be removed inside hot basic blocks. The number of removed spill loads/stores is small and spills are mainly removed by dead registers than unused ones. One can see that only a small percentage of spills in hot blocks are removed. This is mainly due to the fact that profitable live ranges of dead registers tend to be quite short. Although the number of spills removed is small, since they are inside hot basic blocks they may lead to a significant number of dynamic spill loads/stores which could translate to much reduced cache accesses (we show these results later). The main reason for dead register reallocation doing better than unused register reallocation is that too much compensation code cost associated with the latter renders it unprofitable.

Benchmark	Basic Block		Spills in hot basic blocks		
	Hot	Cold	Orig.	Removed By Dead Reg.	Removed By Unused Reg.
Mcf	147	360	38	6	1
Twolf	820	5260	701	36	0
Vpr	223	4620	106	28	8
Parser	751	4898	510	7	0
Bzip2	115	808	121	9	1
Gzip	544	1220	81	1	0
Pegwit	81	1200	25	6	1
Epic	91	1001	54	8	2
G721(en)	94	257	67	9	1
G721(de)	90	260	62	5	1
Adpcm(en)	17	25	13	4	1
Adpcm(de)	12	23	8	0	0
Mpeg2(en)	729	1729	261	8	7

Table 1. Hot/cold basic blocks and spills in hot basic block

Table 3 shows the total number of dead/unused registers inside hot basic blocks and the ones our re-allocation algorithm is able to utilize. From the results, we can see that on average there are 1 to 2 dead registers and 2 to 3 unused registers in hot basic blocks. Again although a high number of unused registers exist, one is rarely able to use them due to high compensation code cost. The dead registers although lesser available one is able to use them better due to low compensation code cost.

Benchmark	Re-allocation Runtime (s)
Mcf	0.225121
Twolf	33.769402
Vpr	12.379912
Parser	6.742614
Bzip2	1.046684
Gzip	1.045277
Pegwit	0.408376
Epic	0.578952
G721(en)	0.229046
G721(de)	0.276383
Adpcm(en)	0.064986
Adpcm(de)	0.034935
Mpeg2(en)	3.604020

Table 2. Re-allocation Runtime

Benchmark	Dead Registers		Unused Registers	
	Total	Used	Total	Used
Mcf	213	5	398	1
Twolf	1287	31	2509	0
Vpr	186	21	305	7
Parser	963	6	1601	0
Bzip2	401	7	1924	1
Gzip	1389	1	1752	0
Pegwit	127	2	222	1
Epic	114	6	175	2
G721(en)	176	7	236	1
G721(de)	150	5	219	1
Adpcm(en)	36	3	51	1
Adpcm(de)	28	0	42	0
Mpeg2(en)	2507	6	4095	6

Table 3. Dead/unused registers

Table 4 shows the number of static spill loads/stores in each benchmark program before and after register re-allocation and the number of dynamic spill loads/stores in each benchmark program before and after register re-allocation. Note that the number of static spill loads/stores is almost the same before and after optimization since the number of spill loads is much larger than the number of spill stores. So in many cases, zero store instructions have to be inserted into cold basic blocks while removing the spill loads in hot regions. On an average, the number of *static spill loads/stores* is reduced by 0.4% actually.

Benchmark	# of Static Spill loads/stores		# of Dynamic Spill loads/stores				Dynamic spills reduction
	Original	After realloc.	Original	After realloc.	Reduction By Dead	Reduction By Unused	
Mcf	217	218	4812685743	3542262843	1166444308	103978592	26.4%
Twolf	5877	5850	30894581907	23905946482	6988635425	0	22.6%
Vpr	3701	3682	19499342512	17489991407	1903457820	105893285	10.3%
Parser	2141	2138	14626637444	13582652855	2043984589	0	7.1%
Bzip2	764	738	10000323963	11330809521	1159018593	60495849	12.2%
Gzip	692	698	18934853875	17230658924	1704194951	0	9.0%
Pegwit	896	892	4285820	3823682	427601	34537	10.8%
Epic	1228	1228	9798175	8543447	1100102	154626	12.8%
G721(en)	138	136	12676763	11723812	946368	6583	7.5%
G721(de)	129	128	12086701	11644174	391578	50949	3.7%
Adpcm(en)	31	28	2253395	1919388	297225	36782	14.8%
Adpcm(de)	15	15	239979	239979	0	0	0.0%
Mpeg2(en)	1603	1607	90038842	87770838	1505421	762583	2.5%

Table 4. Dynamic spills before and after re-allocation

On the other hand, the number of *dynamic spill loads/stores* is reduced significantly. The reduction ranges from 0% to 26.4%. Mcf and Twolf have great dynamic spills reduction since there are both dead registers and spilled variables in hot regions; moreover the allocator could profitably allocate dead registers here removing a large number of spills. But for Mpeg2(en) and Adpcm(de), the reduction is only 2.5% and even 0% due to the almost zero dead registers re-allocated in the in hot regions due to non-profitability. There is a big variation in the results since the benefit brought by register re-allocation is really dependent on the benchmark and dead registers left by the original register allocation scheme and most importantly their profitability.

The results about total dynamic loads/stores before and after register re-allocation are shown in Table 5. The reduction in total dynamic loads/stores depends on the proportion of dynamic spill loads/stores in total dynamic loads/stores for the benchmark. For example, in Mcf, the dynamic spilled loads/stores are reduced by 26.4%. However the removed spill loads/stores are only 7.4% of total dynamic loads/stores since the spill loads/stores are only 28% of the total dynamic loads/stores in the whole program. The main reason for overall low percentage of load.stores is attributable to the excellent base line register allocator.

From the above results, we show that we can reduce the number of dynamic loads/stores in a program which should lead to a better performance due to less data cache misses and also less power consumption due to less activities to data cache. We measured the impact in terms of both performance and power consumption based on a Pentium III 800MHZ processor. The processor is based on 0.18-Micron technology. It has a 16KB, 4-way, 32 bytes per line L1 data cache. Pentium III is a high performance processor; its deep pipeline helps a lot in hiding cache misses (we believe if we apply our method to an in-order improvement in terms of both performance and power). We use Pentium performance monitoring counters [14] to measure the actual cycles spent on each benchmark binary before and after register re-allocation. To make the results as accurate as possible, the OS (Linux) is booted into single user mode. Also, each benchmark binary is executed 10 times and then the average result is reported.

Benchmark	Original	After Realloc.	Dynamic loads/stores reduction
Mcf	17276562150	16006139250	7.4%
Twolf	45802380479	38813745054	15.3%
Vpr	47941788368	45932437263	4.2%
Parser	140486168992	138442184403	0.7%
Bzip2	45545794471	44326280029	2.7%
Gzip	28943908385	27239713434	5.9%
Pegwit	6019701	5557563	7.7%
Epic	25395598	24140870	4.9%
G721(en)	66963592	66010641	1.4%
G721(de)	60527440	60084913	0.7%
Adpcm(en)	1328804	1246537	6.2%
Adpcm(de)	620882	620882	0.0%
Mpeg2(en)	510742100	504874096	0.4%

Table 5. Dynamic loads/stores before and after re-allocation

Benchmark	Original Cycles	Cycles after reallocation	Reduction in cycles
Mcf	893774734092	883454762685	1.15%
Twolf	1607987490372	1416068950276	11.93%
Vpr	433874349698	428680928523	1.20%
Parser	1141834310440	1134944257476	0.60%
Bzip2	333622715307	316351684226	5.2%
Gzip	526584953131	510056316432	3.14%
Pegwit	57110945	55227790	3.30%
Epic	114274358	75441562	33.98%
G721(en)	674386104	664377838	1.48%
G721(de)	629962091	619473971	1.66%
Adpcm(en)	29629061	29166789	1.56%
Adpcm(de)	23769464	23769464	0.00%
Mpeg2(en)	3555239972	3548858134	0.20%

Table 6. Cycle counts

Benchmark	Original energy consumption (J)	after re-allocation (J)	Reduction
Mcf	98.1923	96.1565	2.07%
Twolf	189.0554	165.2092	12.61%
Vpr	80.9813	78.8029	2.69%
Parser	225.1782	222.8119	1.05%
Bzip2	69.5816	66.9382	3.8%
Gzip	73.6566	70.6745	4.05%
Pegwit	0.0104	0.0098	5.43%
Epic	0.0321	0.0274	14.62%
G721(en)	0.1195	0.1178	1.46%
G721(de)	0.1099	0.1086	1.23%
Adpcm(en)	0.0039	0.0038	2.89%
Adpcm(de)	0.0027	0.0027	0.00%
Mpeg2(en)	0.7629	0.7573	0.73%

Table 7. Energy consumption

The impact to the performance in terms of overall execution time is shown on Table 6. The deviation for reduction in cycles is 0.1. The experiments prove that our register re-allocation approach can boost the performance of some benchmarks. For example, the performance of Epic benchmark is improved by about 34%. According to Pentium performance monitoring counter, the average data memory references (L2 misses) without register re-allocation are 51493198, while after optimization, the references are reduced to 32183485. The reason for Epic's good result is that we have removed some spills responsible for a huge number of L2 misses, which were "critical loads/stores". Their removal boosts performance. On the other hand, for many benchmarks, the improvement on the performance is minor. The reason is that the underlying implementation of a Pentium hides a lot of load/store latencies and makes performance improvement difficult for most optimizations. In other words many of the removed load/stores were in fact due to cache hits which had low latency and which could be hidden due to out of order processing. For these benchmarks, there is still a significant benefit in term of data cache power efficiency due to reduced dynamic activities of data cache. Because it is impossible for us to measure data cache power separately in a Pentium processor, we estimate data cache power consumption based on CACTI cache model version 3.2 [8]. CACTI model accepts cache configuration parameters and gives average cache access power. We multiply the number of cache accesses with this power consumption per access number given by CACTI to estimate the dynamic power of D-cache. CACTI does not count leakage power. So we use 10% of the average per access power given by CACTI as an estimate of D-cache leakage power, which is same as Wattch power model [15]. In every cycle, there is D-cache leakage power consumption. But only when D-cache is accessed, there is D-cache dynamic power consumption. Table 7 shows improvement in energy. Due to reduced cache activity one can see that in case of Twolf and Epic there is a significant energy reduction.

Benchmark	Orig. energy-delay product (J.S)	after reallocation (J.S)	Reduction
Mcf	109702.2564	106187.4471	3.20%
Twolf	379998.3830	292434.4382	23.04%
Vpr	43919.6425	42226.6506	3.85%
Parser	321395.1992	316098.8014	1.65%
Bzip2	29017.4838	26470.0044	8.78%
Gzip	48483.0790	45059.9609	7.06%
Pegwit	0.0007	0.0007	8.55%
Epic	0.0046	0.0026	43.63%
G721(en)	0.1007	0.0978	2.92%
G721(de)	0.0866	0.0841	2.88%
Adpcm(en)	0.0001	0.0001	4.41%
Adpcm(de)	0.0001	0.0001	0.00%
Mpeg2(en)	3.3902	3.3596	0.90%

Table 8. Energy-delay product

Table 7 shows the improvement on energy and Table 8 shows the improvement in energy-delay product for data cache. The reduction in energy-delay product ranges from 0% to 43.6%. The best improvement occurs for TWolf and Epic. Both TWolf and Epic improve both in performance as well as in terms of D-cache accesses and thus, their overall energy-delay benefit is high. Bzip, Gzip and Pegwit show moderate improvement in overall energy delay product. Finally the rest show small improvement due to small performance as well as smaller D-cache accesses. These results show that our post-pass register re-allocation approach can improve the energy efficiency of binaries produced by even a

sophisticated register allocator to a good degree without any code growth.

6. RELATED WORK

Our work is focused on profile-guided post-pass register re-allocation to utilize the dead/unused registers available in the binary thus reducing the dynamic spill load/store instructions. The transformed binary is more efficient in terms of both performance and energy consumption. Registers are always a precious resource inside a processor. It is critical to utilize registers efficiently. Chaitin et al. [1] devised an algorithm to represent register allocation as a graph-coloring problem. His allocator becomes the standard and basic graph coloring based register allocator. Briggs et al. [2] developed two improvements, i.e., optimistic coloring and rematerialization, to Chaitin-style graph coloring register allocation. However, Chaitin's allocator and Briggs's allocator do not support live range splitting and have to spill the live range as a whole, which may lead to dead registers in the binary generated. A live range splitting based allocator can alleviate the dead register problem. The standard splitting based allocator is invented by Chow and Hennessy [3]. Later, Bergner et al. [4] proposed interference region spilling which integrates live range splitting into a graph coloring based allocator. However, we have shown in the paper even an allocator supporting live range splitting cannot eliminate dead register problem completely.

Besides our work, there has been other research work tackling dead/unused register problem. For example, Cooper and Simpson [5] developed an algorithm to do splitting directly targeted at the problem of unused registers after allocation. In their work, each time the register allocator needs to spill a value, it checks to see if it would be cheaper to spill the entire live range or split that live range into smaller pieces, some of which will be able to be colored and so will not have to be spilled. Their work is done inside a compiler. In Harvey's master's thesis [6], he devised an algorithm to do local register promotion to address the similar problem we do. His algorithm is less aggressive than Cooper and Simpson's and works as a post-pass like ours. However, his algorithm can only work on single basic blocks. Lu and Cooper [7] also looked at the problem of using unallocated registers to promote values into registers, but they focused on utilizing the results of pointer analysis to determine which scalar variables can be safely kept in registers. Hank [23] devised a region-based compilation technique which repartitions the whole program into regions instead of considering hot regions in a function. Our method is quite different from the above approaches due to several considerations such as compensation code placement and cost issues not addressed by him.. We carefully form hot regions and then model the problem as matching between dead/unused registers considering compensation code costs the key goal being to avoid compensation code on hot paths. Some papers [21, 22] brought out some algorithms for interprocedural register allocater. Their scheme focused on interprocedural levels and splitting registers across calls. David W. Wall [20] delayed the register allocation phase to link time to improve program performance.

Our register allocation not only improves program performance, but also data cache energy efficiency. Most of the work on reducing cache power consumption is focused on architectural perspective. Su and Despain [9] evaluated the effectiveness of a number of low power cache structures. Block (i.e. line) buffering involves latching the last cache line, while sub-banking involves only powering portions of the L1 cache. Ko and Balsara [10] investigated a similar technique that they call Multiple-Divided Modules (MDM). In [11], a small and energy efficient L0 data cache is introduced in order to reduce power consumption of the memory hierarchy. Our method aims to reduce data cache power by reducing the number of dynamic loads/stores. Cilio and Corporaal [12] had a similar idea but they focused on global variable promotion and they assumed that there were dedicated registers for global variables.

7. CONCLUSION

In this work, we proposed a feedback-directed post-pass register re-allocation framework based on profile information to improve the energy efficiency of program binaries. The basic idea is to remove spill loads/stores in hot regions by utilizing dead and unused registers thus reducing dynamic load/store instructions and data cache power consumption. We show that the static code size increase due to our framework is very minimal – in fact a small decrease takes place. Our experiments on SPEC2000 and MediaBench show that our approach always reduces dynamic spills significantly and also improves performance to some extent. Overall, the energy-delay product of the binaries is improved ranging from 0 to 43 % with an average of 7.5%

8. REFERENCES

[1] G.J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. *Register allocation and spilling via graph coloring*. Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, June 1982, pages 98-105.

[2] P. Briggs, K. D. Cooper, and L. Torczon. *Improvements to graph coloring register allocation. ACM TOPLAS*, Vol. 16, No.3, pages 428--455, May 1994.

[3] Fred C. Chow and John L. Hennessy. *The priority-based coloring approach to register allocation*. ACM Transactions on Programming Languages and Systems, October 1990, pages 501-536.

[4] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe, *Spill Code Minimization via Interference Region Spilling*, Proc. of the 1997 ACM SIGPLAN Conf. on PLDI, pp. 287–295. June 1997

[5] Keith D. Cooper, L. Taylor Simpson: *Live Range Splitting in a Graph Coloring Register Allocator*. International Conference on Compiler Construction. Page 174-187.

[6] T. J. Harvey, *Reducing the Impact of Spill Code*, Master's Thesis, Rice University, May 1998.

[7] Keith D. Cooper, John Lu: *Register Promotion in C Programs*. Proc. of the 1997 ACM SIGPLAN Conf. on PLDI : 308-319

[8] Premkishore Shivakumar and Norman P. Jouppi. *CACTI 3.0: An Integrated Cache Timing, Power, and Area Model*. WRL research report 2001/2.

[9] C. Su and A. Despain. *Cache Design Tradeoffs for Power and Performance Optimization: A Case Study*. Proc. of International Symposium on Low Power Design, 1995.

[10] U. Ko, P. T. Balsara, and A. K. Nanda. *Energy Optimization of Multi-Level Processor Cache Architectures*. Proc. of International Symposium on Low Power Design, 1995.

[11] J. Kin, M. Gupta, and W. Mangione-Smith. *The Filter Cache: An Energy Efficient Memory Structure.* IEEE Micro, December 1997.

[12] Andrea G. M. Cilio, Henk Corporaal. *Global Variable Promotion: Using Registers to Reduce Cache Power Dissipation*. CC 2002: 247-260

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. *Mediabench: A tool for evaluating and synthesizing multimedia and communications systems.* International Symposium on Microarchitecture, 330–351, 1997.

[14] Don Heller. Rabbit*: A Performance Counters Library for Intel/AMD Processors and Linux.* http://www.scl.ameslab.gov/Projects/Rabbit/

[15] D. Brooks, V. Tiwari, and M. Martonosi. *Wattch: A framework for architectural-level power analysis and optimizations.* In 27th Annual International Symposium on Computer Architecture, June 2000.

[16] Simon Segars. *Low Power Design Techniques for Microprocessors.* Conference Presentation on IEEE International Solid-State Circuits Conference (ISSCC), Feb. 2001.

[17] Alfred V. Aho, Ravi Sethi and Jefferey D. Ullman. *Compilers, principles, techniques, and tools.* Addison Wesley, 1986.

[18] Mach-Suif Backend Compiler, The Machine-Suif 2.1 compiler documentation set. Harvard University, Sep. 2000, http://ececs.harvard.edu/hube/research/machsuif.html.

[19] L. George and A. Appel. *Iterated Register Coalescing*. ACM Transactions on Programming Languages and Systems, 18(3), May 1996, pp. 300-324

[20] David W. Wall. *Global Register Allocation at Link Time.* Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, 1986.

[21] Steven M. Kurlander Charles N. Fischer. *Minimum Cost Interprocedural Regisster Allocation.* Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1996.

[22] Vutsa Santhanam and Daryl Odnert. *Register Allocation Across Procedure and Module Boundaries*. Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, 1990.

[23] Richard E. Hank. *Region-based compilation*. Ph.D thesis, University of Illinois Urbana-Champaign, 1996.

Appendix

Algorithm For Hot Region Formation

```
hot_region_list = Null;

For each Basic Block B ∈ CFG
  If (B is hot) && if (B ∉ any hot region)
    If (any B's precessor ∉ any hot region)
      && (any B's successor ∉ any hot region)
      create a new hot region HR;
      HR = {B};
      hot_region_list = hot_region_list U HR;
    Else If (B's precessor ∈ hot region HR)
      HR = HR U {B} ;
    Else If (B's successor ∈ hot region HR)
      HR = HR U {B};
    EndIf
  EndIf
EndFor

// merge connected hot regions
change = true;
While (change)
  change = false;
  For each hot region HR_i ∈ hot_region_list
    For each hot region HR_i ∈ hot_region_list
      If ( ∃ B_1 ∈ HR_i) && ( ∃ B_2 ∈ HR_i)
        && (B_1 and B_2 are connected in CFG)
          Hr_i = Hr_i U Hr_i;
          hot_region_list = hot_region_list - Hr_i;
          change = true;
      EndIf
    EndFor
  EndFor
EndWhile
```