Resolving Register Bank Conflicts for a Network Processor

Xiaotong Zhuang Santosh Pande
Georgia Institute of Technology, College of Computing
801 Atlantic Drive, Atlanta, GA, 30332-0280
{xt2000,santosh}@cc.gatech.edu

Abstract

This paper discusses a register bank assignment problem for a popular network processor--Intel's IXP. Due to limited data paths, the network processor has a restriction that the source operands of most ALU instructions must be resident in two different banks. This results in higher register pressure and puts additional burden on the register allocator. The current vendor-provided register allocator leaves the problem to users, leading to poor compilation interface and low quality code.

This paper presents three different approaches for performing register allocation and bank assignment. Bank assignment can be performed before register allocation, can be performed after register allocation or it could be combined with the register allocation. We propose a structure called register conflict graph (RCG) to capture the dual-bank constraints. To further improve the effectiveness of the algorithm, we also propose some enabling transformations.

Our results show the phase ordering of first doing register allocation and then assigning banks can reduce the number of spills with affordable costs of additional instructions.

1. Introduction

The dramatic growth in Internet applications has motivated the need for a specialized category of embedded processors called Network Processors (NPs). NPs have fast processing speed and specialized hardware support for network applications. As the speed of the underlying network keeps increasing (Giga-bits per second etc), and the application requirements put the burden of executing complicated tasks on the network processors (such as the intrusion detection, performance monitoring, routing in server farms etc.), the architectural design and compiler optimization for NPs become more challenging. Normally, simpler designs are adopted for NPs to reduce the delay on the critical data paths resulting in high clocking frequency. On the other hand, the instruction set must provide enough functionality for the applications to fulfill their tasks. Due to the simplicity of the instruction set and architecture design, increased support from compiler optimization is needed to bridge the gap and to yield performance. The compiler optimization for network processor has become an important research topic lately [1][2][3][4]. In particular, register allocation issues are very important since the latency of accessing off-chip DRAM is in tens of cycles and normally results in a context switch to the other threads.

The IXP1200 Network Processor

In this paper, we target the dual-bank register assignment problem for a popular network processor family-Intel's IXP. The IXP network processor [5] works with a very fast processor core. IXP2800 (a recent member of IXP family) can reach 1.4 GHZ clock rate and can process 28 million OC-192 packets per second over SONET. The RISC architecture allows a very concise instruction set and all ALU instructions (including plus, minus, shift, XOR, AND etc) take 1 cycle.

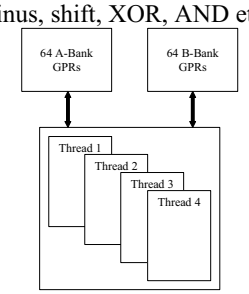


Figure 1 IXP1200 register file structure

Figure 1 shows the block diagram of the register file for the IXP network processor. The register file is divided into bank A and bank B. Four threads share the banked register file. The general purpose registers (GPRs) are physically split into two banks. The ALU unit inside the processor core has two input ports. At any time, either bank A or bank B is connected to one of the two input ports. On the other hand, the output from the ALU unit is accessible from both the banks. This design can potentially support a large number of GPRs and can control the cirtical path delay due to the increased register file connections.

To shorten the execution time for ALU instructions, operands are fetched in parallel. Due to the above design of the data paths coupled with the parallel fetch of operands restrictions are imposed on operand register residency. Each ALU instruction can have two register-resident source operands which must come from the different register banks. For instance, an instruction x=y+z requires register y and z to be in separate banks. However, the destination register x can be in either bank A or bank B.

Dual-bank Register Assignment Problem

The design of dual-bank register file raises the problem of register allocation with consideration to the bank assignments. Without such consideration, register allocation is handicapped, and in some cases, even impossible.

Figure 2 gives two examples to illustrate the dual-bank assignment problem. In these examples, we assume that the total number of physical registers is four, two in each bank. In

Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)
1089-795X/03 $17.00 © 2003 **IEEE**

Figure 2.a, without the above register bank restrictions, each of the variables can be assigned one physical register. However, the three instructions require variable *a* to be in the opposite bank to variable *b*, *c* and *d*. Thus, if variable *a* is in bank A, the other three variables must be in bank B, which is not possible (since there are only two physical registers in bank B). Figure 2.b shows variables *a* and *b* must be in opposite banks. Similarly, variables *a* and *c*, variables *b* and *c* must be in opposite banks too. This creates the problem that even if physical registers are enough, we still cannot satisfy all bank constraints, since the first two instructions require variable *b* and *c* to be in the same bank, which contradicts with the last instruction. Obviously, the first example shows that the dual-bank constraints may cause *imbalance of register* requirements to the two banks. The second example shows that if the bank constraints form *a cyclical conflict*, then no bank assignment is possible without resolving the conflict.

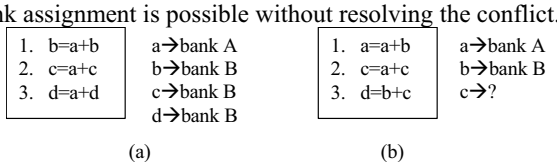


Figure 2 Example of the Dual-bank assignment problem.

The current implementation of the IXP assembler takes passive approaches to the bank assignment problems. The first problem will generate a *"not enough registers"* message to the programmer, while the second problem will cause an error messages prompting the programmer to fix the unresolvable conflict. Obviously, it is difficult for the user to make the right decisions. As we will illustrate later, there are non-obvious trade-offs involved that can actually achieve low-overheads for both register spill and code growth. These tradeoffs are not easily perceivable by the users. With the development of high-level language support for the IXP processor, it is no long appropriate to ask users to resolve the conflicts (and also there is no need to provide "user understandable code" at assembly level), after the code has been transformed from the high-level language. A compiler solution is desirable to automatically assign both registers and banks.

The *dual-bank register allocation problem* is to determine the physical register allocation together with the bank assignment of that physical register for each virtual register. It aims to reduce the overhead due to additional spill code and other extra instructions which can degrade the performance and cause code growth. Speeding up the execution is the first priority; therefore reducing the number of spills becomes most important due to the long memory latency.

Paper Outline

The remainder of the paper is organized as follows. Section 2 describes register conflict subgraph and no-conflict rule, section 3 talks about phase ordering, section 4 proposes the pre-RA approach, section 5 discusses the post-RA approach and section 6 deals with the combined approach. Some enabling techniques are mentioned in section 7. Section 8 shows evaluation results; section 9 discusses related work.

2. Register Conflict subGraph and No-conflict Rule

To represent the register constraints for the dual-bank assignment problem, in this section, we introduce the concept of *Register Conflict subGraph (RCG).*

We build upon the standard representation of *Interference Graph* used in coloring based allocators. *Interference Graph (IG)* represents each *Live Range* as a node in the graph and an edge between two nodes means the two live ranges interfere with each other or are co-live at some program point. We call the edges in the interference graph as *Interference Edges*. To distinguish the interference graph before and after the register allocation, we define *Virtual Interference Graph (VIG)* and *Physical interference graph (PIG)*. The live ranges (nodes) on VIG are *Virtual Live Ranges*, which are associated with virtual registers. Similarly, live ranges on the PIG are *Physical Live Ranges*, which correspond to physical registers [1]. Some edges in the interference graph are further distinguished as *Conflict Edges*, which are defined as follows.

DEFINITION: *Conflict Edge*
If two live ranges interfere in the same ALU instruction as two source operands, the interference edge connecting them is called a conflict edge. They are said to conflict with each other.

Obviously, we have the following claim:
CLAIM: A conflict edge must be an interference edge.

DEFINITION: *Register Conflict subGraph (RCG)*

The register conflict graph is a subgraph of the interference graph consisting only of conflict edges and all nodes.

Similarly, we have the definition of *Virtual Register Conflict subGraph (VRCG)* and *Physical Register Conflict subGraph (PRCG)* based on the underlying interference graph.

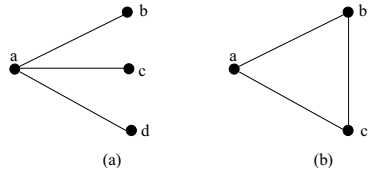


Figure 3 Examples of the RCG.

Figure 3 shows two RCGs corresponding to the examples in Figure 2. Figure 3.a is the RCG for Figure 2.a. Note that all interference edges are conflict edges for this example. We can observe that there are edges from *a* to *b, c* and *d*, which means, {*a*} and {*b,c,d*} form two separate groups that should be put into different banks. In Figure 3.b, the RCG forms an odd-cycle, which means that the nodes cannot be separated into

[1] We define the virtual live ranges to be maximal du-ud chain (or a web) on the VIG, that can be separately allocated. For PIG, physical live ranges correspond to the physical register names, which can be the union of several virtual live ranges that are assigned the same physical register.

Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)
1089-795X/03 $17.00 © 2003 **IEEE**

two groups such that the nodes in the same group do not conflict with each other.

Given a RCG, we can judge if the nodes (either virtual or physical live ranges) in the graph can categorized into two groups with each group being assigned to one of the register banks. If this is possible, we say the RCG is *bank conflict-free*, otherwise it leads to a bank conflict that should be resolved. The bank conflict property is also applicable to the IG, i.e. we say an IG has bank conflict if its RCG exhibits a bank conflict. The problem of determining whether a RCG is conflict-free is equivalent to determining if the RCG is *bipartite*. The following *No-conflict rule* gives the necessary and sufficient condition to judge whether a RCG has conflicts.

No-conflict rule:

The RCG is conflict-free if and only if it contains no odd-length cycle.

Proof: This problem is equivalent to determining if the RCG is a bipartite graph. According to the property of the bipartite graph, no odd-length cycle should exist. This is also a sufficient condition.

Thus, the example in Figure 2.b has a conflict, since its RCG (shown in Figure 3.b) is a length-3 cycle, but the example in Figure 2.a is conflict-free due to the absence of any cycle.

Note that, the No-conflict rule applies to both VRCG and PRCG; however, it gives no guarantee for balancing register assignment in each bank group. From the perspective of register allocation, the number of total physical registers available for each bank is fixed. Typically, the number of physical registers available to bank A equals that of bank B which means register allocator should make an effort towards balancing allocated registers equally between the two banks. We define such a problem *as Balanced Dual-bank Register Assignment*. Accordingly, the RCG with equal number of nodes in each bank group and conflict-free is called *Balanced Conflict-free RCG*. Therefore, the example in Figure 2.a shows a conflict-free but not a balanced conflict-free RCG.

3. Phase ordering

Register bank assignment is closely related to the register allocator that performs virtual to physical register mapping. There are three approaches we can take to perform register allocation together with the bank assignment.

The first method is to assign register banks *before* virtual registers are mapped to the physical registers. We call it *pre-RA bank assignment* approach. All conflicts are resolved on the VRCG before the register allocator takes over.

The second approach is to do register allocation first and then assign the banks to physical registers, at the same time resolve conflicts on the PRCG. This is called *post-RA bank assignment.*

Finally, a combined register allocation approach can consider the register bank assignment at the same time as register allocation.

4. Pre-RA Bank Assignment Approach

We can designate the bank assignment for each virtual register before the register allocation (mapping of virtual to physical registers) is done. According to the No-conflict rule, if there are odd cycles of conflict edges in the VRCG, the VRCG has conflicts. The following claim says that sometimes the conflicts may continue to exist after the register allocation.

Claim: If the VRCG doesn't meet the No-conflict rule, the conflicts will persists after a Chaitin style register allocation assuming no spill is generated.

In general, any graph coloring allocator coupled with other phases meets the above claim. These include, for example, Chaitin's[10], Briggs[11], and Appel & George's [12] register allocators. The claim can be easily verified. During the register allocation, the only possibility is to map *multiple* virtual registers to the *same* physical register through coalescence or coloring. If an odd cycle exists on the VRCG, no edge on the cycle can be collapsed since the interfering nodes cannot be coalesced during the register allocation.

The pre-RA bank assignment approach regards the register allocation pass as a black box and assigns the banks to the virtual registers first. After the bank assignment is done, virtual registers are grouped according to their bank assignments. The register allocation is then done separately in for each bank.

To avoid the conflicts, all odd-length cycles should be removed to make the graph bipartite. A straightforward way to remove odd-length cycles is to break the cycle at some node point in the RCG by splitting the live range of the node.

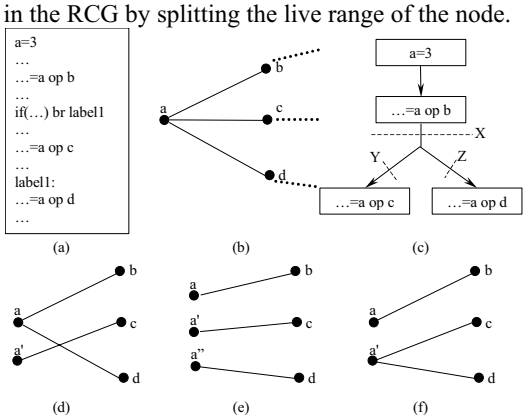


Figure 4 Edge breaking through live range splitting on RCG.

Figure 4 shows an example of breaking a cycle in the RCG. Figure 4.a gives the code segment in which the live range of variable *a* consists of 1 definition and 3 uses across a conditional branch. Figure 4.b shows the RCG. Variable *a* conflicts with *b, c* and *d*. Figure 4.c shows the live range using the control flow graph (i.e. du/ud chain of variable *a*). In Figure 4.b, if we want to break the cycles passing the edge (a,c), (because (a,c) is a part of an odd cycle) we can simply split the live range at point Y (Figure 4.c), which means after point Y, the live range is renamed, for example to *a'*. Figure 4.d shows the RCG after splitting. At the split point, we need to insert a move instruction *a'=a*, so the live range gets

Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)
1089-795X/03 $17.00 © 2003 **IEEE**

separated. One instruction is added in the code and one extra node is added on the RCG. If we want to break the cycles spanning both edges (a,c) and (a,d), one choice is to split at both points Y and Z, i.e. renaming the live range to *a'* after point Y and renaming the live range to *a"* after point Z (Figure 4.e). This requires two move instructions and two additional nodes on the RCG. However, we can split the live range at point X, which leads to the RCG in Figure 4.f. We rename the live range at point X to *a'*. In this case, the cost is only one move insertion and one additional node on the RCG but we cannot break the cycles that pass both (a, c) and (a,d).

The problem of making RCG bipartite (breaking all odd cycles) with minimal cost is shown to be NP-complete (please refer to Appendix A for details) by reducing a graph problem called *Maximal Bipartite Subgraph* [7] to it. There has been substantial work done on the Maximal Bipartite Subgraph problem in graph theory [8][9]. Several approximation algorithms have been proposed. For example, [8] gives an algorithm with complexity of $O(n^4)$. However, these approaches assume that the underlying graph does not have length-3 cycles which are not true in our case. Besides, they target the minimal number of edges removed to make the graph bipartite, while we must consider live range splitting on the nodes and the cost is not uniform for node splitting. Finally, their analysis gives a rather loose lower bound with regard to the quality of the solution. Actually the solution of our algorithm is far beyond the lower bound calculated by these papers. Specifically, our heuristic algorithm below takes into consideration the special properties of RCG that can greatly reduce the complexity of the algorithm while maintaining quality of the solution.

Before presenting the heuristic algorithm, we briefly discuss the detection of odd cycles in the RCG. Here, we define that if two cycles have at least one edge that is different, then the two cycles are said to be *different*. For any algorithm that can resolve all the conflicts in the RCG, it needs to break all the odd-length cycles in the graph. However, a simple estimation tells us that finding all odd-length cycles will take exponential time to finish. A brute-force algorithm must try up to $\sum_{k=1}^{(n-1)/2} C_n^{2k+1}$ possibilities to find out all odd cycles, where n is the total number of nodes in the RCG. This sum has a complexity of $O(2^n)$, since we know $\sum_{k=0}^{n} C_n^k = 2^n$, which is almost twice of $\sum_{k=1}^{(n-1)/2} C_n^{2k+1}$.

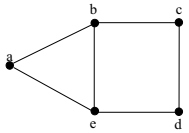


Figure 5 Duplication in counting the odd cycles.

However, in real programs most of the odd cycles are small (Please refer to section 8). Another observation is that the brute force searching may include duplications. In Figure 5, two odd cycles can be found i.e. abe and abcde. However, if edge (a,b) is removed, both cycles are broken. If edge (b,e) is removed, then cycle abcde still remains. This example shows that if two cycles share edges, breaking one of them can cause the other to disappear automatically.

Our heuristic algorithm starts with the shortest odd cycles, after breaking shorter ones, we move on to find longer cycles and break them, until all cycles are gone. As shown in Figure 5, the removal of shorter cycles may break longer cycles as well. The main data structure for odd cycle detection is called *Breadth-First Hierarchy* described below.

4.1 Breadth-First Hierarchy

Given a RCG, we build a *breadth-first hierarchy* from one of the root nodes following the breadth-first searching algorithm. The procedure is as follows:

```
Input: root r
Output: The breadth-first hierarchy
Level_num: number of levels in the hierarchy;
Node_level_set[]:array of sets;
Algorithm:
Function Build_Breadth_First_Hierarchy(node r)
Begin
  Node_level_set[1]←{r}; mark r;
  Level_num=1;
  While Node_level_set[Level_num] not empty do
    For each node p in Node_level_set[Level_num] do
      Add all p's directly connected neighbors that have not
      been marked to Node_level_set[Level_num+1];
      Mark all newly added nodes.
    od
    Level_num++
  Endw
  Return Level_num, Node_level_set[]
End
```

Figure 6 Building the breadth-first hierarchy.

The root node r is the only first level node. After visiting the root node, we visit and mark its directly connected neighbors which are marked as the second level nodes. Next we visit neighbors of second level nodes that are unmarked and assign them level three and so on. It is easy to notice that the level of a node is its minimal distance to the root plus one. Besides, the time complexity to construct the breadth-first hierarchy is $O(n^2)$. Next, we give two lemmas about the breadth-first hierarchy.

Lemma 1:

If an edge e connects node p and node q on the RCG, then |node_level(p)-node_level(q)|<=1, where node_level gives the level of the node on the breadth first hierarchy.

Proof: Without loss of generality, assume node_level(p) >node_level(q), if there is an edge from node q to node p, then p should be on level node_level(q) +1, which proves the Lemma.

Lemma 2:

The RCG is conflict-free, iff any edge (p,q) => |node_level(p)-node_level(q)|=1.

Proof: From Lemma 1, we only need to show node_level(p)<>node_level(q). If this condition is not satisfied, (i.e. if node_level(p)=node_level(q)), we find a path from root r to p called path(r,p) and a path from r to q called path(r,q), each with node_level(p) nodes. Let s be the latest common node for the two paths, i.e. path(s, p) and path(s,q)

Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)
1089-795X/03 $17.00 © 2003 **IEEE**

only share node s. We then have an odd cycle s->p->q->s. On the contrary, we can separate the nodes into two groups; all nodes in odd level form one group, and all nodes in even level form another group. Then there are no edges within these two groups and thus, the graph is bipartite and conflict-free.

Lemma 1 shows that the edges on the RCG only appear between nodes from adjacent levels on the hierarchy. Lemma 2 implies that no two nodes on the same level being connected is equivelant to the RCG being conflict-free. In other words, if a RCG is not conflict-free, there must be an edge that connects two nodes on the same level. We call such an edge *Parallel Edge*. Next, we present a lemma that will be used later in our heuristic algorithm to detect odd cycles with length k, where k is an odd number greater than 3. This lemma can help to build a fast algorithm to detect all such cycles.

Lemma 3:

The smallest odd cycle is of length k (k is an odd number), iff there is no parallel edge on any breadth-first hierarchy up to level (k-1)/2.

Proof: Briefly, the proof is similar to that of lemma 2. If there is a parallel edge on level less than (k-1)/2, we will find an odd cycle that is less than k.

Thus, to find all the odd cycles of length k (assuming non-existence of shorter odd cycles) in the RCG, we build n breadth-first hierarchies with each of the n nodes as root nodes. All the hierarchies only need to be built up to (k-1)/2 level and checked for parallel edges. Since the algorithm runs faster when k is small and most odd cycles are actually small, we find this odd cycle detection algorithm finishes quickly in our implementation.

4.2 Live Range Splitting Patterns

Live range splitting patterns represent the possibilities a live range can be split. For each live range, we can find out all splitting patterns. For example, in Figure 4, we observe 4 possibilities to split the live range, i.e. X, Y, Z or (Y, Z). For each splitting pattern, we can calculate its cost. Although the number of splitting pattern may grow exponentially, in practice only a few live ranges contain a large number of uses as source operands. This is probably due to the nature of the applications running on the network processors. Also, the live range defined as connected du/ud chains can achieve value separation, which leads to smaller number of splitting patterns due to reduced number of uses associated with each live range. In implementation, we specify a limit of 1000 patterns for each live range, otherwise the live range is forced to be split as if it is involved in every cycle. In our evaluation, we show that this almost invariably happens.

4.3 The Pre-Register Bank Assignment Heuristic Algorithm

As mentioned before, our heuristic algorithm breaks odd cycles from the shortest, i.e. size three and goes to longer cycles as it proceeds. The algorithm is listed in Figure 7. It executes several iterations each one breaking all cycles of length m. m takes odd integers from 3 to n. During each round, two sets are built. The *Cycle_set* stores all cycles with length m. The *Pattern_set* stores all patterns. Then we examine each pattern to see how many cycles it can break in the *Cycle_set*, the priority function for applying a pattern is calculated as the number of cycles it can break divided by the cost (the number of moves inserted). The pattern with highest priority is chosen. After a pattern is applied to CFG and VRCG, all broken cycles are removed from *Cycle_set* and the *Pattern_set* is also updated, since new live ranges are added and the old live ranges might be altered. The algorithm picks the most favorable pattern and proceeds with that pattern. The following claim guarantees the algorithm will eventually remove all odd cycles.

Claim: Assume k to be an odd number greater than 3, then after all length<k odd cycles are broken, the breaking of length-k cycles will not create shorter odd cycles.

This is obvious, since during the whole process, nodes are not merged to form new cycles on RCG. The complexity of the algorithm in the main loops is roughly O(n*P*M), where n is the number of nodes on the original graph, P is the maximal number of patterns in the *Pattern_set* and M is the maximal number of the *Cycle_set* size. Since most odd-cycles are short and breaking shorter cycles may break longer ones as well, the outer-most for loop in Figure 7 normally finishes early. From Figure 7, we notice that the update of *Cycle_set* and *Pattern_set* can be done with marginal computation.

```
Input: VRCG, CFG
Output: VRCG (no conflict),CFG, set of split live ranges

Algorithm:
Function Pre_RA_Bank_Assignment
Begin
  Construct patterns (calculate costs) for all live ranges, store to Pattern_set
  For m=3 to n, step 2
    Detect all odd cycles with cycle length m, store to Cycle_set;
    while (Cycle_set <> empty) do
      For each pattern p in Pattern_set do
        w=cost of using p
        bn=number of cycle p can break in Cycle_set
        The priority of pattern p is bn/w
      od
      The pattern with highest priority is applied on VRCG, CFG
      Remove broken cycles from Cycle_set
      Update Pattern_set if necessary
    Od
  Endfor
  Return VRCG, CFG and the set of split live ranges
End
```

Figure 7 Pre-RA bank assignment heuristics.

The drawback of this approach is the difficulty to control the register pressure in each group, which may lead to imbalanced pressure between the two banks during the register allocation. For example, assume that the virtual registers are grouped equally when they are passed to the register allocator. However, it then turns out that one of the groups needs more physical registers to avoid spilling, while the other group has free registers. As it is hard to judge the physical register and spill code that will be generated before the register allocation, the pre-register allocation approach may result in imbalanced spill. In other words, it may increase the overall spill cost. However, after the RCG becomes conflict-free, this problem can be alleviated by making the RCG near-balanced before passing it to the register allocator.

Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)
1089-795X/03 $17.00 © 2003 **IEEE**

4.4 Near-Balancing the RCG

After the live range splitting, it is quite likely that the RCG is no longer a connected graph. By identifying the connected components of the RCG, we can near-balance the number of nodes in the two banks through separate bank assignment to each connected components of the RCG.

Suppose the RCG has m connected subgraphs: G_1 G_2..G_m, each with a subset of the nodes and edges of the RCG. Since the m subgraphs are all conflict-free, i.e. bipartite, we can separate each G_i into GA_i and GB_i, such that no conflict edge is inside GA_i and GB_i (This can be done to construct a breadth-first hierarchy and separate odd level and even level nodes). Let A_i=| GA_i |, B_i=| GB_i| and the number of total nodes is $n = \sum A_i + \sum B_i$. We want to minimize $\left| \frac{n}{2} - \sum C_i \right|$, where C_i=A_i or B_i. In our implementation, we apply exhaustive search, which takes $O(2^p)$, since p is typically less than 10. For larger p, a fully polynomial time algorithm can be derived from the *subset sum algorithm* [6], which closely approximates the optimum in polynomial time.

5. Post-RA Bank Assignment

Although the pre-RA bank assignment can avoid conflicts during the register allocation, it creates imbalanced register requirements (higher chromatic number for one of the register banks). On the contrary, post-RA bank assignment approach allocates register with well-known register allocation algorithms to minimize the spills in the first place. Our bank assignment algorithm is invoked to resolve bank conflicts and balance physical register distribution across banks. The post-RA bank assignment algorithm will not increase spill code. Although some physical live ranges are split after moves are inserted, the cost is much lower than spills.

As we know, the register allocator can map different virtual registers to the same physical register. Therefore, physical live ranges are typically larger than virtual ones. The post-RA bank assignment problem shares many properties with the pre-RA bank assignment problem. Therefore, some of the techniques can be borrowed. However, there are clear differences between these two approaches. Firstly, we cannot simply rename a live range, because each physical live range is allocated a physical register. If a live range is split, we must find an available physical register to hold the new live range. Secondly, the PRCG must be balanced and conflicts removed.

5.1 Cost for Splitting Patterns

The cost of splitting patterns for physical live ranges are calculated differently. Especially the cost to split at a certain point is not just equal to the inserted move instruction. In building the *Pattern_set* for the heuristic algorithm, we categorize the cost for a pattern into 3 types:

1. If the register pressure in the renamed program segment (for example, in Figure 4.c, from point Y to instruction a op c) is less than the number of available physical registers, then splitting cost is set to be the move insertion cost.
2. If condition in (1) is not true, we look around near the splitting point to find the chance of *rematerialization*, and calculate the cost accordingly.
3. Finally, we count the cost of doing *in-place bank exchange*.

5.1.1 Rematerialization

Rematerialization has been used by [11] to free a register through recomputing the value in-place before it is needed to avoid carrying the value in the register. We check for rematerialization before we analyze the splitting patterns; thus the register pressure in some region of the program can be reduced.

5.1.2 In-place Bank Exchange

After all the above endeavors fail, we have the last resort to solve the bank conflicts using this technique. *In-place bank exchange* requires no additional registers, however it requires 4 ALU instructions to remove one conflict edge on the RCG. Although it can be expensive in terms of space in contrast to a register spill, in-place bank exchange saves runtime cycles and most importantly, guarantees the odd cycles can be broken in the worst case without incurring spills.

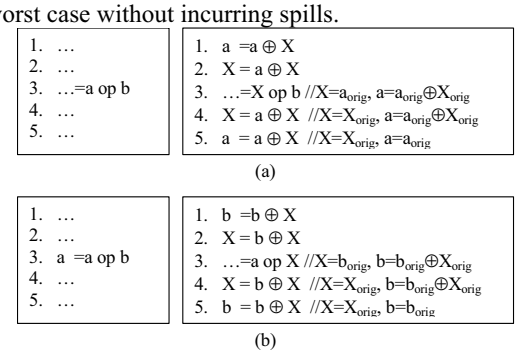


Figure 8 In-place bank exchange.

Figure 8 illustrates two cases to break a conflict edge between live range *a* and *b*. In Figure 8.a, we assume that the destination operand of instruction (3) is not *a*. We insert two XOR instructions before and two XOR instructions after the ALU instruction in line 3. The register X is an occupied physical register that is in the opposite bank to variable *a*, thus it is also in the opposite bank to variable *b*. The first two exchanges put the variable a in X and then the instruction in line 3 is conflict-free. The last two XOR instructions restore the values of *a* and *X*. The conflict edge between *a* and *b* can be removed, because the two variables can be assigned physical registers in the same bank now. Figure 8.b shows the case when *a* is the destination operand. We can exchange *b* and *X* to remove the conflict. In-place bank exchanges are special splitting patterns that are provided for all edges. Therefore all odd cycles can be broken in the worst case with these patterns.

5.2 Balancing Register Numbers in Two Banks

In this section, we discuss the balancing of registers in the two banks. After the removal of all odd cycles, we can apply the same bank assignment approach as the pre-RA algorithm to obtain a near-balanced RCG. After that, we have to reassign some of the live ranges to the opposite bank, which

Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)
1089-795X/03 $17.00 © 2003 **IEEE**

could induce conflicts. The induced conflicts have to be resolved at the cost of inserted instructions.

Suppose the live ranges on the RCG have been grouped into two groups, *BankA* and *BankB*. Also assume, |*BankA*|>|*BankB*|. We attempt to pick one of the nodes in *BankA* and move it to *BankB* and estimate the cost of resolving the conflicts due to this move by again using the minimal cost splitting pattern. This procedure is repeated until the number of nodes in the two banks are equal. If the difference between |*BankA*| and |*BankB*| is small (as it usually is), we can attempt all combinations of moves for (|*BankA*|-|*BankB*|)/2 nodes from *BankA* to *BankB*, which gives better solution than moving nodes one by one from *BankA* to *BankB*, but takes slightly longer time to finish.

```
Input: PRCG, CFG
Output: PRCG (balanced and conflictless), CFG

Algorithm:
Function Post_RA_Bank_Assignment
Begin
  Register_allocation
  Construct patterns for all live ranges, including in-place exchange
    patterns, store to Pattern_set
  Foreach pattern p in Pattern_set do
    Calculate the cost for p (with available register, rematerialization or
      In-place exchange)
  od

  Break all odd cycles //the same as in Pre_RA_Bank_Assignment

  Near-balancing the two bank groups
  If |BankA|<>|BankB| then
    Balance the bank by moving nodes between them.
  Endif

  Return PRCG, CFG
End
```

Figure 9. Post-RA bank assignment heuristics.

5.3 Heuristic Algorithm for Post-RA Bank Assignment

We discuss the algorithm for post-RA bank assignment algorithm in this section. In Figure 9, the procedure Post_RA_Bank_Assignment consists of 4 parts: register allocation, constructing and calculating the cost for pattern, breaking odd cycles and balancing the two bank groups. After register allocation is done (assuming a monolithic register file), the *Pattern_set* is constructed. Cost calculation is more complicated than the pre-RA bank assignment. Next, odd cycles are broken as in the previous section. Bank balancing has two steps. The near-balancing algorithm in the previous section is applied first, because it incurs no cost. Then, the approach described in section 5.2 is applied to achieve a perfect balance.

6. Combined Register Allocation Approach

As mentioned before, it is possible to combine register allocation with register bank assignment. However, as many register allocation algorithms have been proposed in literature, we do not want to delve into all the possibilities. In this section, we briefly introduce our combined algorithm with Briggs-style register allocation algorithm [11]. In contrast to the original algorithm, we have several modifications.

The allocator takes nodes from the interference graph and pushes them to one of the stacks.

1. Two stacks are maintained for the two banks.
2. During "coalesce", coalescence is performed only when the two nodes do not interfere with each other. In addition, coalescence should not create new odd cycles.
3. In the "simplify" stage, we push each node on the IG to one of the stacks. Nodes can be marked as "spill" or "conflict". Nodes are pushed in the following order.
   a) We pick a node and push it to a stack that causes no conflict (with nodes still on the IG) and no spilling (with neighbor number less than the number of registers in one bank).
   b) If a) fails, find a node that does not need spilling (but has bank conflicts) and with minimal cost to resolve the conflict, push it to one of the stacks.
   c) If both a) and b) fail, find a node that must be spilled but with minimal spill cost as calculated by Brigg's algorithm and push it to one of the stacks.
4. During "select", we pop nodes from the two stacks one by one in the order they are pushed to the stacks.
   a. Give the node a color (the color must belong to the bank of that node) that is different from all colored neighbors on the IG, and that has no conflict with the nodes already on the IG.
   b. If the color is available, but there is conflict, we resolve the conflict as in the previous section.
   c. If both a) and b) fail, the node must be spilled.

One difficulty in the combined approach is that we do not know the bank assignments of the remaining nodes on the IG during "simplify". In the worst case, we have to assume all neighbors will be in the same bank, so only half of the registers are available for coloring. This causes a lot of nodes to be marked as "spill". Possible improvements to this problem are being investigated.

7. Some Enabling Techniques

This section discusses two types of optimizations that can help with the aforementioned approaches. As shown later, these approaches do not require additional virtual or physical register but can help to reduce the number of odd cycles on the RCG.

7.1 Removal of Length-3 Odd Cycle Conflicts

Figure 10 shows an example on how to remove conflicts involving odd cycles of length three. Recall that we attempt to break odd cycles in the order of increasing lengths (refer to Figure 7 and Figure 9) and thus, applying this transformation could break cycles of higher order as well. Figure 10.a is a code segment where three operands s1, s2, s3 form an odd cycle of length 3 on the RCG. Note that, all the three operands are live in but not live out, while the 3 destination registers are the only live-outs. Figure 10.a shows the code transformation that removes the conflict edge <s1,s3>. Also notice that the instruction in line 3 is supported by the IXP, which does a left shift before minus. Figure 10.c gives a general form of the code segment and the rule for this transformation to be legal. f1 and f2 must be supported instructions. "plus", "minus" are

Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)
1089-795X/03 $17.00 © 2003 **IEEE**

most commonly seen operators that find this transformation useful. This optimization is applicable to pre-RA and post-RA bank assignment. But for post-RA bank assignment, it merges live ranges, which may result in new odd cycles. In our implementation we have a simple check it before the code transformation is applied in post-RA setting.

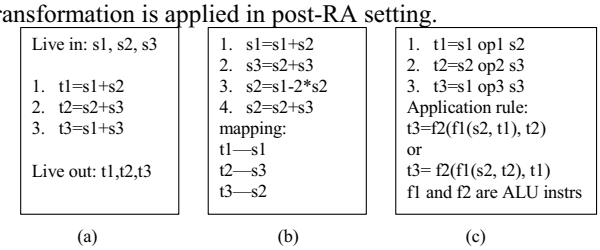


Figure 10. Example for removal of conflict involving triangle cycles.

7.2 Application of Algebraic Laws

Algebraic laws such as associativity, distributivity can be applied to change the edge connectivities on the RCG, so as to reduce conflicts on the graph. These optimizations can be invoked on the RCG to break some of the cycles.

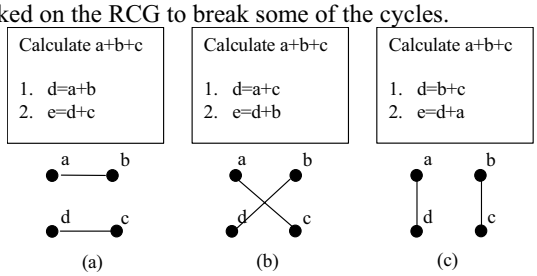


Figure 11. Example for application of algebraic laws.

In Figure 11, three cases are shown to calculate a+b+c. With associativity, we can calculate a+b first or a+c first or b+c first. Therefore, on the RCG, 3 kinds of connectivities are possible, given that all ALU instruction can have at most two register operands. The choice largely depends on the number of odd cycles each one would create. In our implementation, we focus on the number of triangles that can go through these edges. It can be easily counted using breadth-first hierarchy based on the first two levels of nodes. The calculation order with the least number of length-3 odd cycles is chosen. More generally, most 2-operand ALU instructions satisfy associativity can be transformed with this law.

8. Experimental Results

We evaluate the algorithms with the Intel-provided IXP1200 Developer Workbench 2.01. The IXP1200 workbench supports cycle-accurate simulation for IXP microengines and other peripherals with high fidelity. It provides both assembler and a C compiler supporting a subset of ANSI C.

We experiment with 8 benchmark programs to see the effectiveness of the three approaches. These benchmarks are collected from Commbench[16], Netbench[17], and a packet scheduling algorithm from [18]. The benchmark programs are rewritten in IXP C code and a few of them are directly written in assembly (micro-code). For the assembly code generated by the C compiler, we restore the virtual registers. Figure 12 shows the flowchart of the compilation process. The register allocation and bank assignment pass have three modules, i.e. Post-RA, Pre-RA and combined-RA. The general register alloator is the one proposed by Briggs et.al. [13]. Our pass builds the CFG, IG and RCG from the assembly code, after simple translation of the assembly directives. The IXP assembly consists of only 40 RISC instructions, which makes the translation easy. For one thread, the number of total physical registers is 32 (the benchmarks are assumed to be run on only one thread). Therefore, 16 registers are available in each bank.

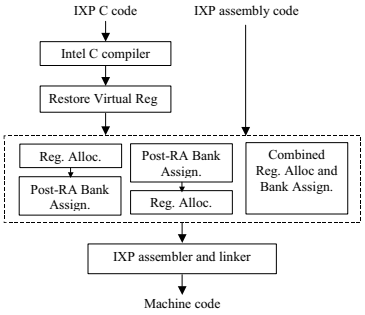


Figure 12 Compilation flowchart.

Table 1 shows the properties of the benchmark programs. The code size is the number of instructions after code generation. The number of live ranges and interference edges are listed in 3rd and 4th column. On average, the degree of the live ranges on the interference graph is about 9. The last column shows the number of conflict edges. Conflict edges are much less than interference edges. Only a small fraction of instructions become conflict edges, because only ALU instructions with two source GPR operands establish conflict edges in the RCG. Among these instructions, some conflict edges are identical.

Table 2 shows the cycle length distribution. We separate columns into two categories. Column 2 to 4 show the distribution of cycle length before the two enabling techniques in section 7 are applied. Column 5 to 7 are the distribution after these techniques are used. We apply the enabling techniques before register allocation and bank assignment. From Table 2, we find most cycles are of length 3. Cycles with length greater than or equal to 7 are rare. The techniques in section 7 seem to have limited effects, especially for larger benchmarks.

Table 3 gives the number of instructions inserted to apply the patterns (we do not included instructions for spills, this will be counted in the next table) to break odd cycles and balance the banks. They include “move insertion” and “in-place exchange” etc. for post-RA bank assignments. The results signify that post-RA adds more instructions than the other two. This is due to the more ambitious conflict breaking attempted by this stage adding many instructions. Since virtual registers have been allocated physical registers, more odd cycles may result. Nodes on the physical RCG are more costly to split, because they represent several nodes for virtual live ranges. A combined-RA approach tends to generate fewer additional instructions, however, as we will see later, more spills are created.

Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)
1089-795X/03 $17.00 © 2003 **IEEE**

Table 4 gives the number of spills generated by each approach. The combined-RA is worst, since the graph coloring works poorly when two stacks are assumed. Many nodes are marked as spill when pushing to the stack because the number of neighbors they have on the graph is larger than the number of physical registers in one register bank, but actually their neighbors on the graph may finally go to the opposite bank, which is not known at the time they are pushed onto the stack. Post-RA is the best in reducing number of spills (since it assumes one bank when doing RA), which compensates the increased number of additional instructions due to the high cost of spills.

Table 5 compares 4 of the 8 benchmarks for runtime performance. In summary, pre-RA and Combined-RA are very close, while post-RA is about 7% (ranging from 6% to 9%) better than Combined-RA.

The compilation time for all benchmarks is within 1 second on a Pentium 4 machine. Obviously, the combined-RA approach is polynomial time algorithm. For pre-RA and post-RA algorithm, the majority of the compilation time is spent on cycle breaking. As mentioned in section 4.3, the complexity of the cycle breaking is O(n*P*M). If the outmost loop can finish early, the complexity is close to O(PM). Since we have set the bound for M, the complexity is further controlled by O(P)*Max(M). Finally, P is the maximum among the numbers of different odd-length cycles. Normally, this is the number of length-3 cycles on the RCG, which should be in $O(n^3)$.

In conclusion, our Post-RA bank assignment is successful in breaking odd cycles and bank balancing without increasing spills. The extra instructions cannot offset the benefits of spill reduction. Pre-RA generates more spills than Post-RA but less than the current version of the combined-RA.

Table 1 Benchmark applications.

	Code Size	#live ranges	#interference edges	#conflict edges
Drr	108	11	55	25
Fir2dim	447	36	120	71
Frag	271	26	133	65
Kmp	123	13	53	27
Lzw	126	18	105	36
Md5	913	142	630	246
Wraps (receive)	875	145	643	236
Wraps (send)	921	135	464	193

Table 2. Cycle length distribution.

	Without algebraic law & triangle conflict removal			With algebraic law & triangle conflict removal		
	Length-3	Length-5	Length ≥ 7	Length-3	Length-5	Length ≥ 7
Drr	7	1	0	7	1	0
Fir2dim	48	2	0	48	2	0
Frag	51	3	1	49	3	1
Kmp	8	0	0	8	0	0
Lzw	49	4	0	44	4	0
Md5	836	13	2	827	12	1
Wraps (receive)	1132	19	3	1101	17	3
Wraps (send)	842	10	0	840	10	0

Table 3. Comparison for number of inserted instructions.

	Pre-RA	Post-RA	Combined-RA
Drr	3	5	5
Fir2dim	10	18	11
Frag	8	20	8
Kmp	3	8	4
Lzw	4	10	3
Md5	38	59	19
Wraps (receive)	35	62	23
Wraps (send)	29	55	21

Table 4. Comparison for number of spills.

	Pre-RA	Post-RA	Combined-RA
Drr	2	0	4
Fir2dim	5	0	7
Frag	3	0	8
Kmp	2	0	2
Lzw	2	0	5
Md5	30	23	35
Wraps (receive)	45	38	56
Wraps (send)	52	38	57

Table 5. Comparison for runtime cycles.

	Pre-RA	Post-RA	Combined-RA
Drr	205391	188910	198910
Fir2dim	147149	134982	142812
Frag	26760	25281	26351
Kmp	146909	137829	143618

9. Related Works

Although this paper studies optimizations for a special network processor architecture with dual-bank register file, partitioned register file has long been adopted by many commercial DSPs such as Texas Instruments' VLIW chips. The IXP network processor's register file differs from theirs in that 1) only one function unit; 2) the parallel access to the register file is restricted to the two source operands.

A recent architecture paper [13] studies multi-banked architecture and shows performance advantage. [14] talks about the register allocation for VLIW machines. However, none of these papers deal with the issue of dual bank constraints discussed by us. [15] studies the register allocation problem for a dual-bank register file. Register access requires both register number and a bank specifier, which are determined by a control register. Since their architecture does not require source operands to be in different banks, the approaches used are different from us.

Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)
1089-795X/03 $17.00 © 2003 **IEEE**

[4] models the register constraints on IXP as an integer linear programming problem, which leads to excessive compilation time (compilation time of several seconds is reported in their paper for relatively small benchmarks which makes it unacceptable as a standard compilation pass). In addition, we believe our solution being light-weight in terms of compilation time is scalable towards the future generation of IXP processors with code store size at least quadrupled (IXP2400 vs IXP 1200). Secondly, the ILP formulation only guarantees optimality in stage one, however, the overall solution is still sub-optimal. Specifically, in the presence of spills, the solution generated by their method may not be optimal. Finally, their paper does not include transformations such as in-place exchange, which trades code size for less number of spills (sometimes solely reduces the number of spills) which enhance overall quality of solution over what exhaustive methods can discover in a non-transformed space .

REFERENCES

[1] J. Wagner and R. Leupers, "C Compiler Design for an Industrial Network Processor", *LCTES*, June 2001.

[2] J. Kim, S. Jung, Y. Park, "Experience with a Retargetable Compiler for a Commercial Network Processor", *CASES'02*, 2002.

[3] J. Liu, T. Kong, and F. Chow, "Effective compilation support for variable instruction set architecture", *In Proc. PACT'02,* Sep. 2002.

[4] Lal George, Matthias Blume, "Taming the IXP Network Processor", *PLDI'03*, pp.26-37,2003.

[5] "IXP 1200 Network Processor: Programmer's Reference Manual", Part No. 278304-010. Dec. 2001.

[6] T.H.Cormen, C.E.Leiserson, R.L.Rivest, *Introduction to Algorithms*, MIT Press, 1989.

[7] C. H. Papadimitriou and M. Yannakakis. "Optimization, Approximation and Complexity Classes", *Journal of Computer and System Sciences*,Vol.43,pp.425-440,1991.

[8] S. Poljak and Z. Tuza, "Bipartite subgraphs of triangle-free graphs", *SIAM J. Discrete Math*. Vol. 7, pp.307-313, 1994.

[9] J.A.Bondy, S.C.Locke, "Largest bipartite subgraphs in triangle-free graphs with maximum degree three", *Journal of Graph Theory*, Vol.10, pp.477-504, 1986.

[10] G.J. Chaitin, "Register allocation and spilling via graph coloring", *In Proc. of the SIGPLAN Symposium on Compiler Construction*, pp. 98-105, 1982.

[11] P.Briggs, K.Cooper, L. Torczon, "Improvements to Graph Coloring Register Allocation", *ACM TOPLAS*, 16(3), May 1994, pp. 428-455.

[12] L. George and A. Appel. "Iterated Register Coalescing", *ACM Trans. on Prog. Lang. and Systems*, 18(3), pp.300-324, May 1996.

[13] J. Cruz, A. Gonzalez, M.Valero, N.P. Topham, "Multi-plebanked register file architectures", *ISCA*, Jun. 2000.

[14] S. Jang, S.Carr, P.Sweany, D.Kuras, "A Code Generation Framework for VLIW Architectures with Partitioned Register Files", *In Proc. of the 3rd Int'l Conference on Massively Parallel Computing Systems*, 1998.

[15] J.Park, J.Lee, S.Moon, "Register Allocation for Banked Register File", *LCTES 2001*, Jun. 2001.

[16] T. Wolf and M. Franklin, "CommBench – A Telecommunication Benchmark for Network Processors", *ISPASS,* 2000.

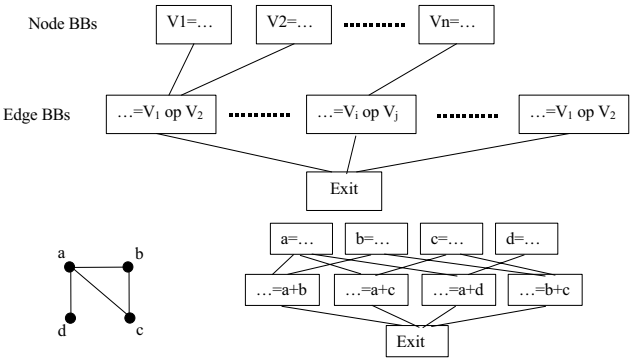
[17] Memik, G., W.H. Mangione-Smith, and W. Hu., "NetBench: A Benchmarking Suite for Network Processors", *ICCAD,* pp. 39-42, Nov. 2001.

[18] Xiaotong Zhuang, Jian Liu, "WRAPS Scheduling and Its Efficient Implementation on Network Processors", *HiPC 2002*, pp. 252-263, 2002.

Appendix A

The problem of making RCG bipartite (break all odd cycles) with minimal cost is NP-complete.

Proof: Firstly, it is trivial to show the problem is polynomial-time verifiable. Next, we reduce the maximal bipartite subgraph problem to it. The maximal bipartite subgraph problem is to find the minimal number of edges to be deleted to make a given graph bipartite. Suppose, we are given an instance of this problem—an undirected graph G(V,E). We construct a program code with two-level CFG. The first level basic blocks (BBs) represent nodes on G called node BBs. The second level BBs represent edges on G called edge BBs. Each edge BB is connected to the two node BBs on the edge. In each node BB, there is one instruction that makes an assignment to a variable. In each edge BB, the two variables from its node BBs conflict as source operands. Now, the constructed graph has a RCG equivalent to G. Breaking an edge on G is equivalent to splitting the live range in the corresponding edge BB by inserting a move instruction before the ALU instruction. Therefore, we have reduced the maximal bipartite subgraph problem to the problem of making RCG bipartite with minimal cost. The following shows an example.



Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)
1089-795X/03 $17.00 © 2003 **IEEE**