ON THE FLY CLIQUE PARTITIONING FOR REGISTER ALLOCATION

Ali Sianati  
Department of Electrical and Computer Engineering  
Shahid Beheshti University  
Tehran, Iran

Rasoul Saneifard  
Department of Engineering Technology  
College of Science and Technology  
Texas Southern University  
Houston, Texas 77004  
Saneifard rx@tsu.edu

Masoud Abbaspour  
Department of Electrical and Computer Engineering  
Shahid Beheshti University  
Tehran, Iran

ABSTRACT

In this endeavor a novel approach to a register allocation algorithm for Digital Synthesis is presented. Register allocation and functional unit allocation can reduce the overall cost of Application Specific Integrated Circuits (ASICs). Clique partitioning is one of the most efficient methods to assign variables to registers while minimizing the total count of the registers. On the Fly Clique Partitioning for Register Allocation (OCPRA) attempts to construct cliques on the fly during the lifetime phase of the register allocation. OCPRA utilizes a lemma which is only true for a scheduled Control Data Flow Graph (CDFG) and constructs cliques in each of its scheduled cycles.

Key Words: register allocation, digital synthesis, clique partitioning

1. INTRODUCTION

Register allocation is the process of assigning variables to a set of registers in order to synchronize the register transfers. Register optimization is one of the optimizations in digital synthesis. Register optimization which refers to the task of assigning operands to registers while reducing the total count of the registers. The lower the registers’ count, the less wiring and multiplexing is necessary, therefore resulting in cost reduction. The most difficult part of this process is solving loops and branches in which two or more basic blocks share the same register, while only one of them is operational at any given moment.

Several methods are used for register allocation, such as clique partitioning, graph coloring, Integer Linear Programming (ILP), etc. Clique partitioning and coloring are similar, as clique partitioning performs on the compatible graph of the operands, while coloring is used on the conflict graph of the operands, and these graphs complement each other [10]. These two methods are widely used while their being NP-complete or NP-hard make them unsuitable for large CDFGs. Tseng’s FACET method finds the compatible graph by means of variables’ lifetimes and allocates the register by clique partitioning 0. Bridge method utilizes clique partitioning which is NP-complete (difficult problems in non-deterministic polynomial time) 0. Stock presented a procedure that applies a conflict graph and allocates the register by coloring but it is NP-complete. Furthermore, the new register transfers in this method are needed for loops 0. Paulin’s method, Hardware Allocator (HAL) uses weighted clique partitioning which is NP-hard 0.

Nam-sun Woo uses sets of variables for register allocation, but their method does not support branches 0. Kundahi in his Register Allocation (REAL) method utilizes lifetime and leftedge algorithms which take \( O(n^2) \) and solves the branch problem by coloring, but do not support loops 0. Wang’s Global Register Allocation Optimal Algorithm (GRAOA) uses slices and solves the branch problem by maximum weight matching, which produces optimal results that are similar to REAL. Its complexity is \( O(t^3) \) where \( t \) represents the
number of slices, and $k$ denotes the maximum weight $0$. Balakrishnan uses ILP and register files which make his method NP-hard $0$.

2. LEMMA
A lemma which is the basis for this algorithm will be discussed and proved before introducing the algorithm.

Lemma: In a scheduled CDFG, if variable $x$ is compatible with variable $w$ and variable $w$ is compatible with variable $z$ in the order of the cycles visited, then $x$ is compatible with $z$.

Proof: In a scheduled CDFG, when moving upward from the last cycle to the first to find the lifetimes, if $x$ is compatible with $w$, it shows that $x$ is either not alive or is in another basic block. The same thing exists for $w$ and $z$, so $z$ is alive when $x$ is not alive. In the cyclic order, $x$ is compatible with $z$.

This lemma is only true on the scheduled CDFG. Since an interval graph may consist of many compatibles without considering the time, this lemma may not work properly in some instances [9].

3. ALGORITHM STRUCTURE
The algorithm herein receives its ability from clique partitioning in graph theory, pruning in networks, insertion sorting and hashing. Also, it locates lifetimes as other algorithms do, but during each cycle, the cliques and clique partitions are reconstructed.

This algorithm utilizes some properties of compatible graphs so that each node (operands in an operation) can initialize and develop numerous cliques. Out of the resulting cliques, the largest are selected in order to obtain the maximum number of clique partitions.

Some structures are used here for simplicity and better understanding of the use of programming language. In the following sections, these structures are discussed.

3.1 Clique
As mentioned above, all cliques made by a node named Initial Node or Head Node are obtained; however, the clique structure retains the initial node of the clique. Along the initial node, all the operands of the same variable, during the lifetime of the head node, are retained by each clique and are used for binding. Each clique has a link list named Levels which holds compatible nodes with Head Node. Each cell of this link list is another structure named “Level” and each compatible node is part of one Level.

A list of special nodes, added to the clique but not used, is retained to reduce the search time for these types of nodes. The list is entitled “Broken Nodes” (Sec 3.1.2).

Quantity of a clique is coordional of the clique which is the number of levels (Sec 3.1.1) in a clique plus one. Another number is held by each clique to indicate the number of levels which are empty to reduce the quantity of the clique.

3.1.1 Level
Level is a structure which holds the nodes that are compatible with the head of the clique, but not with each other. Each head can use one of these nodes to make the maximum clique.

This structure is a hash table of conflicting nodes which increases the speed and decreases the search time between conflicting nodes. Each Level refers to the preceding Level in the next clique, based on the algorithm. Each two levels referring to each other are alike, based on the lemma. This reference makes the process of pruning easier, as it eliminates further search for levels.

3.1.2 Level Node
Each level in the clique contains nodes with conflict, so a structure for these nodes is needed. This structure contains the operand which starts a lifetime and therefore, a clique. To construct the clique, a list of compatible LevelNodes must be retained; therefore, this structure contains a pointer to the next Level in the list of levels, as each LevelNode is compatible with all LevelNodes in the referenced Level.

If a LevelNode has no reference to one of the next Levels in the levels list, it is held in the Broken Nodes of the clique.

3.2 Conflict List
During the process of finding the lifetimes $0$, a list of intervals that overlap (conflict) is developed in each cycle. These intervals exhibit when a value is generated as an output of an
operation and the last time the variable is referenced as an input to another operation. Hence, each BasicBlock (linear sequence of operation codes having one entry point and one exit point) uses a hash table of Conflict objects from previous cycles of the BasicBlock, and another table for new conflicts found in each cycle as it processes that cycle. This list is used for finding lifetimes and also for moving along the BasicBlocks and Control Blocks.

3.2.1 Conflict
This structure is a link list of nodes (operands) of the same variable that are in different Control Blocks. This list solves the problems of Control Blocks (i.e., branching and looping) because all the intervals made by a variable from different blocks are gathered and processed together.

4. Algorithm
BindCDFG(CDFG)
{
    cur_block = lastBlock(CDFG); //find last block to traverse CDFG from bottom to top
    Queue Q_Blocks; // a queue for level traverse of the CDFG
    Add cur_block to the Q_Blocks;
    while(Q_Blocks is not empty)
    {
        cur = remove from Q_Blocks ;
        For each BasicBlock that jump into cur
        Add to Q_Blocks if BasicBlock is not visited before;
        FindCliques(cur,clique_list);
    }
    Register regs; //a set of registers to be bound
    while clique_list contains any Clique
    {
        newReg=add a new register to regs;
        clique=Select first clique from clique_list; // first clique is always max in list
        for each operand in the cur //head and correspondent
        bind newReg to Selected_Node and Operands in the head of the corresponding Clique to the Operand of the Selected_Node;
        /* Prune Selected_Node by using prune reference in the level */
        bool remove=true;
        prune_chain_level=prune reference of level;
        while (prune_chain_level is not null)
        {
            remove Selected_Node from prune_chain_level;
            if (remove == true)
            {
                if prune_chain_level has no more Level Node
                Remove prune_chain_level from it's clique;
                else
                remove=false;
            }
            else if prune_chain_level has no more Level Node
            Increase the Reduction Counter of clique;
            prune_chain_level=prune reference of prune_chain_level;
            // end of pruning
        }
        // end of for each in line 16
        // end of while in line 10
        //function for processing each basic block
    }
    FindCliques(BasicBlock BB, cliques_list)
    {
        for each Basic Block that BB jumps to it
        Copy live variables of Basic Block to BB's live variables;
        Loop from last cycle of BB to first cycle
        {
            for each operation scheduled in this cycle
            {
                for each Input operand in current cycle
                if operand is alive, Find the Conflict in BB's live variables which is the same as operand and add operand to all the Cliques that are made of the variables of this clique ; // continue the life time
                else Add a new Conflict made by this operand and add it to the list of new conflicts;
            }
        }
    }
}
39: if list of new conflicts of BB contains any conflict then
40: AddClique(cliques_list, live vars of the BB);
41: } // end of loop in line 33
42: Move Conflicts which are beginning an interval to live vars of BB
43: for each Output operand in predecessor cycle if operand is alive, find the Conflict in BB's live variables which is the same as operand and add operand to all the Cliques that are made of the variables of this clique, then remove the conflict //end the life time
44: Mark set BB as bound;

} // AddClique (Cliques_list, livevars of bb)

57: } //end of else in line 50
58: } // end of for each in line 47
59: for each Bubble_Clique in the Bubbling_list
60: Remove the Bubble_Clique from its index and add it to the end of the Cliques_List
61: add a new clique for each conflict in the list of new conflicts and add them to end of Cliques_List

It is obvious that this algorithm goes through several steps to construct all possible cliques by means of the first lemma.

4.1 BindCDFG

This function gets a scheduled CDFG as its input and binds registers to it. In the first line of the code the last Basic Block is attained to start traversing CDFG from bottom to top to find lifetimes. To find the lifetimes in each BasicBlock, the CDFG must be traversed backwards so the last BasicBlock is the first Block to get visited.

Each BasicBlock is visited when all of the successor BasicBlocks are previously visited, so either Level First Traversing or Depth First Traversing are used.

After constructing all cliques by accessing FindClique for each BasicBlock, the registers are bound. A loop on all cliques accomplishes this goal. To assign a register to a clique, the clique must be maximized. The first clique in the cliques_list is the maximum clique, hence the top clique is selected and the nodes found for it. A new register is added to the registers list and head, and all relative operands are bound to this register.

Each Level contains one of the nodes that participates in the maximum clique, and the LevelNode refers to the next Level in the levels list. The operand of this LevelNode is bound to the new register and must be removed (pruned) from the cliques_list. Referring the present Level to the next level in another clique simplifies this task. While levels which are
accessed by this reference have a link to another Level, the task of pruning continues. Each time a Level is accessed and the selected LevelNode is removed, the task continues to the next Level (if any). This task is similar to pruning in some networking algorithms. During the pruning operation, if a previous level has no more LevelNode(s) and that Level (in the same condition) has been removed, it can be omitted from the clique, which reduces its quantity. This action helps to find cliques with intersections while their Levels refer to each other. After pruning the Operand of the selected LevelNode, the corresponding clique is removed from the cliques_list.

4.2 FindClique

In this function, BasicBlock and the cliques_list are inputs, and new cliques are constructed from BasicBlock. Furthermore, this function loops through the cycles of BasicBlock and finds lifetimes and related conflicts.

In the first two lines of the code (lines 33-34), live variables of successor blocks are moved to this block. Each live variable is introduced by a conflict in the successor block. If a variable is alive in more than one block, the operand of the conflict is added to the related conflict in the current block. Adding the same variables from different blocks solves the problem of control blocks. To avoid replication, each operand has a unique identity, so during the movement of the live variables, operands with the same identity are added only one time.

If a variable is alive (contained in the Conflict List) and used as an output operand, it must be removed from the list of conflicts in order to indicate the end of its lifetime. Beforehand, this operand is added to all the cliques with the same live variable during the lifetime of the cliques. The cliques can be identified by the Conflict related to this variable.

If a variable is alive and used as an input, no further action must be performed. However, the operand must be added to cliques related to it as previously mentioned.

If a variable is not alive and is used as output, it is skipped because it is no longer useful after this cycle. Thus, register space is saved.

If a variable is not alive and it is input, it starts a new lifetime. A new conflict is made by the operand and is added to the new conflict list of the BasicBlock.

4.3 AddClique

After finding all the lifetimes in each cycle, this function is called and then takes the cliques_list and livevars of the BasicBlock as input.

This function loops through all cliques created up to this cycle and attempts to expand their size. A lemma is introduced and works to assist the algorithm.

During this process, the function takes only one of the conflicts in the new Conflicts List, as other conflicts have the same condition as the selected conflict. Each Clique Head and last Level are examined to check their relationships with the selected conflict.

If the Head is not compatible with the selected conflict, the related clique is added to the bubbled_Clique.

If the Head is compatible with the selected conflict, but none of the LevelNodes in the last level are compatible with it, then it and all other conflicts on the new Conflicts List are added to the previous level, as they cannot increase the size of the clique. Furthermore, the conflicts in the new Conflicts List are added to the Broken Nodes list, and the clique in this condition is added to the bubbled_Clique list.

If the Head is compatible with the selected conflict, they can form a clique. If one of the LevelNodes in last Level is compatible with the selected conflict (and other conflicts in the new Conflicts List), a new level is added to the clique, and then all conflicts in the new Conflict List are added to this level. Then, the reference to the next Level of Broken Nodes that are compatible with the selected conflict is set to the new Level. If this clique is not the first clique in this condition, the pruning reference of the last clique is set to the new Level. LevelNodes composed of these new conflicts are added to the Broken Nodes of the clique.

After examining each clique which has increased in size, the clique must be bubbled and sorted. The term “bubble” is given to this
process because cliques that have not grown are selected to find their index on the new clique list. Finding the new index of each bubbled clique is accomplished by using an insertion sorting process. After growing each clique (if any), the quantity of bubbled cliques are compared to the quantity of grown cliques. If the size of the compared clique is larger than or equal to the grown clique, it is removed and added prior to the grown clique (it is added before and not after the grown clique, because the moved clique may have a Level reference to the grown clique). The order of cliques must be maintained so that the order of Level references is similar to the order of cliques on the list.

5. LOOPS

Solving the loop problem is more complex, because lifetimes in the loop are circular (i.e., intervals at the end of the loop are fed back to the beginning of the loop). To overcome this problem, the lifetimes in the loop are found so that the process of clique-building may continue.

To find the lifetimes in a loop, a simple rule is considered. If the intervals found at the end of a Basic Block are moved back to the beginning, the found lifetimes at the end of this process stay put if the task is repeated. This rule may be used for loops, and only once, move the lifetimes at the end of the loop to its beginning to find cliques for this loop.

6. COMPLEXITY

To explain the complexity of this algorithm, two cases are considered and described below. In the first case, it is assumed that all operands in the CDFG are compatible, and in the other, that all are incompatible.

6.1 All compatible

In this case, all operands are compatible (the least clique processing is required), only one Level and LevelNode have to be processed for each clique, so finding the lifetimes and cliques takes $O(n^2)$ for insertion sorting, and uses $O(n)$ for binding.

6.2 All incompatible

In this instance, all operands conflict with each other, therefore, it is necessary to process the most cliques. This requires $O(n^2)$ for finding and $O(n)$ for binding cliques. This case produces the most registers for ASICS.

To better understand the complexity of this algorithm, the actual situation is examined. The number of cliques is equal to the intervals in the CDFG, and, the processing power for each clique is at least equal to the register count. It is assumed that the number of intervals is $n$ and the number of registers bound for the CDFG is $r$, the complexity of the algorithm is $O(n^2)$, and if all the intervals are considered to be incompatible with each other, the algorithm complexity reaches $O(n^3)$.

7. EXPERIMENTAL RESULTS

OCPRA has been simulated utilizing Microsoft® C# (C Sharp) language and Windows operating system with Net framework 1.1 and later. Figure 3 illustrates the CDFG for the differential equation in Figure 10. The output was generated in less than 1/10th of a second on a 2.4 GHZ Intel® Pentium processor. The register allocation for this example is shown in Figure 2. Note that variable $o$ in BB2 is not a declared variable, but is an intermediate value produced by a subtract operation and is later reused in the same operation. Each operand in Figure 3 is indicated by a rectangle containing the variable and the ID for allocation.

```c
int a, dx, x, u, y;
while( x < a ) {
    int xl=x+dx;
    int ul=u-5*x*dx - 3*y*dx;
    int yl=y+dx;  
x=xl;
    y=yl;
    u=ul;
}
```

Figure 1: Differential equation for experimental result

```plaintext
Register 1: 60, 26, 20, 31, 39, 42, 56, 40, 54, 49, 53
Register 2: 55, 24, 13, 57, 6
Register 3: 50
Register 4: 51
Register 5: 41, 80
Register 6: 58, 2
Register 7: 47, 37, 22, 13, 4
Register 8: 44, 35, 10
Register 9: 46, 17, 22, 8
Register 10: 53, 14
```

Figure 2: The register allocation result
8. CONCLUSION

Register allocation is one of the main processes necessary to reduce the overall cost of ASICS. Clique partitioning is one method used so far in this area, but it is complex. OCPRA constructs cliques on the fly while finding the lifetimes of variables. It utilizes a lemma introduced in Section 3. This method performs in low latency in CPU time if hash tables are extensively used and performed correctly.

9. REFERENCES


Transactions on Programming Languages and Systems, Vol. 21, No. 5, September 1999, pp. 895-913.

Ali Sianati received his Bachelor of Science in Electrical Engineering degree from Shahid Bahonar University of Kerman, Iran, in 2005. He received a Master of Science Degree in Computer Engineering from Shahid Beheshti University of Tehran, Iran. He currently works as a project manager for Hacoupian Industries supervising software, programming, and database projects. Also, he is a network administrator and has developed several computer programs similar to a network simulator, NS.Net. Furthermore, he is a member of IEEE. His research interests include networking, P2P networks and C# developing.

Rasoul Saneifard received his BSEE and MSE degrees from Prairie View A & M University, Prairie View, Texas, in 1988 and 1990 respectively, and his Ph.D. in Electrical Engineering from New Mexico State University in 1994, and is a Registered Professional Engineer in the State of Texas. He is currently Associate Professor in the Department of Engineering Technologies at Texas Southern University. He has authored numerous refereed papers that have been published in distinguished professional journals such as IEEE Transactions and ASEE’s Journal of Engineering Technology. He is a member of IEEE, ASEE, Tau Alpha Pi, and is the founder of Students Mentoring Students Association (SMSA). He served as Chair of the Engineering Technology Division of the Conference on Industry and Education Collaboration (CIEC 2010), a division of ASEE. His research interests include fuzzy logic, electric power systems analysis, electric machinery, and power distribution.

Maghsoud Abbaspour received his B.Sc., M.Sc. and Ph.D. from University of Tehran, Tehran, Iran in 1992, 1995 and 2003 respectively. He has joined Computer Engineering department, Shahid Beheshti University, Tehran, Iran in 2005. His research interests include multimedia on wireless network, multimedia on peer to peer network and Network security.