Resource Spackling: A Framework for Integrating Register Allocation in Local and Global Schedulers\(^1\)

David A. Berson, Rajiv Gupta, and Mary Lou Soffa

Computer Science Department, University of Pittsburgh, Pittsburgh, Pa. 15260

Keyword Codes: D.1.3; C.1.3
Keywords: Instruction-level parallelism; Speculative code motion; VLIW, Superscalar

Abstract: We present Resource Spackling, a framework for integrating register allocation and instruction scheduling that is based on a Measure and Reduce paradigm. The technique measures the resource requirements of a program and uses the measurements to distribute code for better resource allocation. The technique is applicable to the allocation of different types of resources. A program's resource requirements for both register and functional unit resources are first measured using a unified representation. These measurements are used to find areas where resources are either under or over utilized, called resource holes and excessive sets, respectively. Conditions are determined for increasing resource utilization in the resource holes. These conditions are applicable to both local and global code motion.

1 Introduction

A variety of local and global scheduling techniques have been developed for exploiting instruction level parallelism. The degree of parallelism in the schedule is affected by register allocation, which is applied either before or after scheduling. Instruction scheduling, which allocates functional units, and register allocation have competing goals. The goal of register allocation is to avoid spills, which tends to result in a few values being held in registers for a long time. The goal of instruction scheduling is to keep all functional units busy, typically requiring a large number of values to be available for future operations. Thus an improvement in the availability of one resource may reduce the availability of the other resource, and possibly result in poor overall quality of generated code.

We present a framework for integrating instruction scheduling and register allocation that is based upon a Measure and Reduce paradigm. Resource requirements are measured and better resource utilization in both local and global scheduling is achieved by moving instructions from areas with over utilized resources to areas with under utilized resources. Integrated allocation of registers and functional units is achieved by allowing simultaneous consideration of the demand for both types of resources during scheduling.

Previous work on local schedulers has treated register allocation and instruction scheduling as separate phases [BEH91, GoH88, Pin93, SwB90]. In addition, the instruction scheduling phases have been based on list scheduling. The separation of phases and use of list scheduling limit the direct assessment of the impact of register and functional unit allocation.

\(^1\)Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-91090809 to the University of Pittsburgh
allocation decisions on the availability of other resources and hence on the length of the schedule. Recent work has incorporated parallel live range information into register allocation, but still separates register allocation from instruction scheduling [Pin93]. The approach we present unifies the allocation of registers and functional units in a single phase and is able to consider the impact of an allocation decision on other instructions.

Work on global scheduling has identified blocks to which an instruction can be moved [AiN88, BeR91, Fis81, GuS90, SHL92] by concentrating on functional unit constraints [EbN89, MGS92] in carrying out the code motion. Only recently has work on global scheduling begun to consider register allocation as a part of the problem [MoE92, NiG93]. Although Moon and Ebcioglu [MoE92] added register constraints, they are only able to exploit unused registers at the beginning or end of a basic block. Our framework allows all idle resources in a basic block to be identified, regardless of when they are idle, and exploited when instructions are available. The framework can be used in conjunction with commonly used methods for performing code motion, such as Trace Scheduling [Fis81], Percolation Scheduling [AiN88], and Region Scheduling [GuS90].

The Resource Spacking framework computes resource requirements for a program. A unified representation of functional unit and register uses is constructed to identify all resource uses that can temporally share the same instance of a resource. Using this representation we compute the maximum number of each resource required at each point in the program. Maximum functional unit requirements correspond to maximum parallelism, while maximum register requirements correspond to the maximum number of simultaneously live values. The resource requirements measures are then used to identify two sets: excessive sets, which are sets of instructions that may be executed concurrently but require more resources than are available, and resource holes, which are areas where a resource is underutilized. Properties are identified for holes and resources, which indicate how instructions can be inserted into the holes to increase resource utilization. Moving an instruction is only beneficial if it can be placed in a hole where all necessary resources are available. The technique is called Resource Spacking due to the process of identifying and filling resource holes.

2 Determining Resource Requirements using Allocation Chains

This section summarizes the measurement of resource requirements used to locate excessive sets and resource holes [BGS93]. Measurement of resource demands depends on the usage characteristics of the resource. The two major types of resources considered in this work, functional units and registers, have different use properties. If a resource is in use only during the execution of an instruction, we say it is a non-spanning resource. If the use of a resource begins during the execution of one instruction and ends during the execution of a subsequent instruction we say the resource is spanning. The instruction that begins the use is called the defining instruction and the instruction that ends the use is called the killing instruction. Functional units are non-spanning resources, while registers are spanning resources.

The resource usage information is represented as a ReuseR DAG, and is computed from the program DAG. Functional unit and register Reuse DAGs are denoted as ReuseFU DAG and ReuseReg DAG respectively. The term Reuse comes from the property that for any edge (A, B) in the ReuseR DAG, where A and B are nodes representing instructions in the program DAG, B can always safely reuse A’s instance of R.

The program DAG in Figure 1(b) is a ReuseFU DAG. A corresponding ReuseReg DAG is shown in Figure 1(c). The selection of which use kills a value must be performed carefully so that the number of simultaneously live values is maximized. To maximize the number of simultaneously live values for the ReuseReg DAG in Figure 1(c) the following
selections have been made: B kills A, I kills H, and E kills both C and D.

Resource requirements for resource \( R \) are measured from the partial order represented by the \( \text{Reuse}_R \) DAG by finding sets of instructions that can reuse the same resource instance, called allocation chains. Formally, a chain is a subset of elements in a partial order such that all elements in the chain are related, i.e., ordered. A decomposition of a partial order into chains is a set of chains such that all elements in the partial order are in exactly one of the chains. A decomposition is minimal if there is no other decomposition of the partial order that has fewer chains. Since the allocation chains are found on the \( \text{Reuse}_R \) DAG, all instructions on one allocation chain can be assigned the same instance of the resource. Each of the sets of nodes \( \{A, C, E, G, H, I, K, L\} \), \( \{D, F, J\} \), and \( \{B\} \) is a chain, and this set of chains forms a minimal decomposition of the DAG. Similarly, the \( \text{Reuse}_{\text{Reg}} \) DAG can be minimally decomposed into the chains \( \{A, B, L\} \), \( \{C, F, G, H, I, K\} \), \( \{D\} \), and \( \{E, J\} \).

The maximum number of independent elements in a partial order is equal to the number of chains in a minimal decomposition[Di150]. Since the partial order is constructed from resource reuse information, the number of chains in a minimal decomposition represents the maximum number of instructions that can execute concurrently and simultaneously live values for \( \text{Reuse}_{\text{FU}} \) DAGs and \( \text{Reuse}_{\text{Reg}} \) DAGs respectively. Thus, the block in Figure 1(a) requires three functional units and four registers to exploit all of its parallelism. A minimum decomposition of a partial order can be found by using a straightforward transformation to a bipartite graph matching problem[FoF65].

### 3 Resource Holes

Resource holes and their properties are located by analyzing the allocation chains for the resource of interest. Given a hole \( h \), the size of \( h \), \( \text{size}_h \), is the number of cycles for which the hole’s resource is available for allocation. \( \text{EAT}_h \), the earliest available time of hole \( h \), is the earliest time that the resource can be allocated to an instruction. \( \text{LAT}_h \), the latest available time of hole \( h \), is the last time that the resource can be allocated to an instruction. These properties are determined by the time of execution of the instructions surrounding the holes. The scheduling of instruction \( i \) in the program DAG is limited by the precedence constraints to a time frame in which it can execute. The time frame

A: load a
B: \( b = 2 * a \)
C: \( c = a + 1 \)
D: \( d = a - 3 \)
E: \( e = c * d \)
F: \( f = c - d \)
G: \( g = e / f \)
H: \( h = g + 5 \)
I: \( i = h * 2 \)
J: \( j = h + 4 \)
K: \( k = i / j \)
L: \( l = b + k \)
is delimited by the instruction’s earliest start time, \( EST_i \), and latest finish time, \( LFT_i \). Let \( \tau_i \) denote the execution time for instruction \( i \). Then \( i \)'s latest start time, \( LST_i \), is given by \( LST_i = LFT_i - \tau_i \). The \textit{slack time} for scheduling instruction \( i \) is given by \( \text{slack}_i = LST_i - EST_i \). The identification of resource holes is performed by examining the instructions’ \( EST \)s and \( LFT \)s on each allocation chain and recording the information for each hole.

Resource holes can occur in two different situations. The first, a free hole, occurs when an instance of a resource is unused in a section of a basic block. Free holes can occur because no instructions from an allocation chain can execute in this section. They can also occur at the beginning or end of a block before maximum demands are encountered.

\textbf{Definition 1} If two consecutive instructions, \( i_j \) and \( i_{j+1} \), on an allocation chain cannot be executed consecutively, i.e., \( LFT_{i_j} < EST_{i_{j+1}} \), then there is a \textit{free hole}, \( h \), such that \( EAT_h = LFT_{i_j}, LAT_h = EST_{i_{j+1}} \), and \( \text{size}_h = LAT_h - EAT_h \).

We refer to the pair \( (EAT_h, LAT_h) \) as the \textit{range} of hole \( h \). As an example, assume the DAG in Figure 1(b) and the chains mentioned earlier, and that all instructions require unit time. The DAG requires eight time units to execute. A free functional unit hole exists between instructions \( F \) and \( J \), with size 2 and range \((3, 5)\). Thus, two instructions could be allocated to that allocation chain between \( F \) and \( J \).

The second type of hole, a slack hole, occurs when resources that are already allocated may be temporally shared. If \( \text{slack}_i = 0 \) then \( i \) is on a critical path and has no flexibility for scheduling without increasing the execution time of the basic block. If \( i \) is not on a critical path then there is some flexibility on when it can be scheduled. Thus, its resources may be available for allocation to another instruction.

\textbf{Definition 2} If there is a set of consecutive instructions \( I = \{i_1, i_2, \ldots, i_n\} \) and a constant \( s \) such that \( \forall i_j \in I \), \( \text{slack}_i = s \), there is a \textit{slack hole}, \( h \), such that \( EAT_h = EST_{i_1}, LAT_h = LFT_{i_n} \), and \( \text{size}_h = s \).

In Figure 1(b) there is a slack functional unit hole involving instructions \( A, B \), and the end of the block. \( B \) has a slack time of 5, and a range of \((1, 7)\). Thus, five instructions could be allocated to \( B \)'s allocation chain. Any number of these five instructions can be allocated between \( A \) and \( B \), with the remainder after \( B \).

Instructions are inserted in holes to avoid increasing the critical path length. Thus a hole must be at least as large as the instructions being inserted in it. There are cases when additional instructions must be placed in the hole to manage the use of registers. In a register slack hole there are instructions which are already using the register. Spill code must be placed in the hole around the inserted instructions to free this register for the inserted instructions. Additionally, the final value computed by the inserted instructions must sometimes be spilled. The following theorem formalizes the conditions under which a set of instructions can be inserted in a hole. The values \( \tau_{\text{store}} \) and \( \tau_{\text{load}} \) are the number of cycles required to store and load a value respectively.

\textbf{Theorem 1} Let \( I \) be a set of instructions, with execution time \( \tau_I \), that requires resource \( R \). During insertion of \( I \) in \( h \) there will not be any increase in the length of the critical path of the basic block containing \( h \), due to unavailability of resource \( R \), if \( h \) satisfies one of the following conditions.

1. \( R \) is a non-spanning resource or \( R \) is a spanning resource and inserting \( I \) in \( h \) requires no spills, and \( \text{size}_h \geq \tau_I \).
2. \( R \) is a spanning resource, and inserting \( I \) in \( h \) requires only the final value computed by \( I \) to be spilled and \( \text{size}_h \geq \tau_I + \tau_{\text{store}} \).
3. $R$ is a spanning resource, and inserting $I$ in $h$ requires only a value computed by instructions already in $h$ to be spilled, and $\text{size}_h \geq \tau_T + \tau_{\text{store}} + \tau_{\text{load}}$.

4. $R$ is a spanning resource, inserting $I$ in $h$ requires both a value computed by instructions already in $h$ and the final value computed by $I$ to be spilled and $\text{size}_h \geq \tau_T + 2\tau_{\text{store}} + \tau_{\text{load}}$.

**Proof:** Since the cases are mutually exclusive, each case is proven in turn.

**Case 1** When no spilling is required the hole must be at least as big as the inserted instructions, whose size is $\tau_T$, giving $\text{size}_h \geq \tau_T$.

**Case 2** When the final value computed by the inserted instructions must be spilled, the hole must be at least large enough to hold both the inserted instructions and the store instruction, giving $\text{size}_h \geq \tau_T + \tau_{\text{store}}$.

**Case 3** When a value computed by instructions already in the hole must be spilled, the value must be stored before the inserted instructions and then reloaded following the inserted instructions. Thus the hole must be at least large enough to hold a store and a load in addition to the inserted instructions, giving $\text{size}_h \geq \tau_T + \tau_{\text{store}} + \tau_{\text{load}}$.

**Case 4** This case is a combination of cases 2 and 3. Summing the additional instructions that must be inserted with $I$ gives $\text{size}_h \geq \tau_T + 2\tau_{\text{store}} + \tau_{\text{load}}$.

In addition to the instructions chosen for insertion in the hole, $I$ must also contain any load instructions for any values needed by $I$ which are not already in registers.

In some situations, instructions must be inserted even when there are no holes available for insertion. In these situations the scheduler creates pseudo holes in the block for each resource needed, resulting in an increase in the critical path length. This process of forcing holes into the block, increasing its critical path length, is called **wedged insertion**.

### 4 Local Scheduling and Register Allocation

In the Measure and Reduce paradigm, local resource allocation is performed by introducing sequentiality between instructions whose resource demands exceed available resources. The sequencing places two instructions, which were on separate allocation chains, onto a single allocation chain. The result is that the two instructions are allocated a single instance of the resource and share it temporally. Sequencing must be performed when the number of allocation chains is greater than the number of resource instances available.

**Definition 3** An excessive set $E_R = \{I_1, I_2, ..., I_m\}$ for resource $R$ is a set of instructions such that

1. all instructions are independent, i.e., $\forall i, j \in E_R, i \notin \text{ancestors}(j) \cup \text{descendants}(j)$, and

2. there are excessive requirements, i.e., $m > |R|$.

Note that condition 1 implies that each instruction is on a separate allocation chain.

For every instruction $i$, the allocation chains can be used to find all resources for which $i$ is a member of at least one excessive set of the resource. Sequentialization is then performed by selecting $i$ to be the instruction with excessive uses that has the greatest slack time to be moved to holes. The slack time is used to prioritize the instructions since it indicates flexibility in finding a place to move the instruction. If there is a set of overlapping holes for all resources that $i$ excessively uses within $i$'s execution range, then $i$ can be inserted in those holes without increasing the critical path length.

If there is no set of holes within $i$'s execution range, then an increase in the critical path length is unavoidable. There are two options. First, there may be a set of holes close to $i$'s execution range to which $i$ can be moved. Second, wedged insertion can be performed to create a set of holes for $i$'s excessive uses. The option that minimizes the increase to the critical path length should be selected. The outline for an algorithm that reduces a block by finding or creating holes is given in Figure 2.
Procedure \text{reduce}\_\text{block}( \text{block} )
\{
\text{While block has excessive sets do}
\{
\quad \mathcal{I} = \text{all instructions in all excessive sets for all resources};
\quad \text{select } i \in \mathcal{I} \text{ with maximum slack};
\quad \mathcal{R} = \text{the set of resources that } i \text{ excessively uses};
\quad \text{if } ( \exists \forall \text{ hole } h_r \text{ whose ranges overlap with each other and } 
\quad \text{ } \quad i \text{'s execution range})
\quad \text{holes} = \text{this set of holes};
\quad \text{else}
\quad \{
\quad \quad \text{close} = \text{the set of holes } h_r \text{ s.t. } r \in \mathcal{R} \text{ whose ranges overlap}
\quad \quad \quad \quad \text{and are closest to } i \text{'s execution range};
\quad \quad \text{wedge} = \text{the set of holes created by } \text{wedged insertion} \text{ for } \mathcal{R};
\quad \quad \text{holes} = \text{the set, either close or wedge that minimally increases}
\quad \quad \quad \quad \text{the critical path length of block} ;
\quad \}
\quad \forall \text{place } i \text{ in } h_r \text{ by adding sequentialization edges};
\quad \text{if ( excessive spanning uses remain )}
\quad \quad \text{spill uses between the excessive set and the hole containing } i ;
\quad \text{remove } i \text{ from excessive set information};
\}
\}

Figure 2: Function \text{reduce}\_\text{block}()

As an example, consider the DAG in Figure 3(a). First assume that the target architecture has at least five registers but only three functional units. Then the nodes C, D, F, G, H, and I are all members of at least one functional unit excessive set. Nodes H and I each have a slack time of one. There is a functional unit slack hole around each of H and I, so H’s hole overlaps with I’s execution range. Figure 3(b) shows the result of inserting I in H’s hole. Dashed arrows indicate sequentializing dependences, i.e., dependences due to reuse of resources rather than data values.

Now assume that only four registers are available and G is selected to kill both C’s and D’s values and I is selected to kill E’s value. Then nodes C, D, F, H, and I are in functional unit and register excessive sets. Node H has slack time but there are no register holes in its execution range. Therefore the algorithm must increase the critical path length. There are a functional unit and a register hole available after G executes since it kills two values and only needs one register for itself. Inserting H in the hole following G would increase the critical path by one instruction. Wedged insertion would increase the critical path length more because the pseudo hole must be large enough to spill and reload a value. Therefore the algorithm chooses the hole close to H instead of performing wedged insertion. The resulting DAG is shown in Figure 3(c).

Although the creation of some live values may be delayed by sequencing, the instructions that compute the live values may need input values. These input values remain live from where they are computed to where the excessive instructions are moved. In Figure 3(c) the value computed by H was delayed until there was a register available. However, E’s value remains live until after both H and I execute. In this example it is impossible to reduce the register requirements below four using just sequentialization. When such a situation occurs sequentialization must be combined with register spilling. There are two options for selecting what values to spill. Either the values in the excessive set may be
computed and spilled, or the input values may be spilled. The option selected depends on what holes are available. Computing and spilling the excess values prior to the excessive set requires additional functional unit and register holes, while spilling the input values requires additional functional unit holes to where the values are moved.

Continuing with the above example, assume the same killing instructions and that the target architecture has three registers and two functional units. As before, the nodes C, D, F, H, and I are in an excessive set and only instructions H and I have slack time. Free functional unit and register holes become available after G executes, and another set of free functional unit and register holes become available after J executes. H and I are placed in the free holes. However, the sequentialization would still leave the excessive set {C, D, E, F}. Thus a spill must be performed. To minimize the number of spills, the algorithm spills the input value, E. The resulting DAG is shown in Figure 3(d).

5 Global Scheduling and Register Allocation

The goal of global scheduling is to move instructions from a source block to a destination block to decrease the execution time of the source block by reducing the critical path length in the source block and avoiding increasing the critical path length in the destination block. The instructions moved are called fill instructions since they are inserted in holes in the destination block. Fill instructions may be found in blocks with the same control dependences and in blocks with different control conditions when the architecture supports speculative execution [SHL92] or guarded execution [HsD86], or when code duplication is performed.

Next we describe how existing global code motion techniques [AiN88, Fis81, GuS90] can use the framework to unify functional unit and register allocation, and determine which code motions are beneficial. To realize a benefit, all instructions that are at one end of a DAG and are on a critical path must be moved together; otherwise the critical path length will not be reduced. We call such sets of instructions critical sets. Consider removing nodes from the top of the DAG in Figure 1(b). The first critical set is \{A\}. When A is moved the length of the DAG is reduced by the execution time of A. Then the
Function fill( dest, source )
{
    reduce = 0;
    While dest has holes do
    {
        cs = next set of critical instructions from source;
        Foreach instruction i ∈ cs
        {
            compute EST_i based on i’s dependences in the destination block
            LFT_i = LFT of the last instruction in the destination block
            /* find overlapping resource holes */
            Foreach instruction i ∈ cs, in decreasing order of EST_i
            {
                Forall resources r required by i
                {
                    select holes h_r such that they overlap with the other holes
                    selected and with i
                    if no such holes exist
                    {
                        undo all moves from the current critical set;
                        return reduce; }
                    Insert i into h_r’s allocation chain; }
                Update the hole description information; }
            reduce = reduce + \min_{i ∈ cs} (τ_i ); }
        return reduce;
    }
}

Figure 4: Function fill()

next critical set is \{C, D\}. Note that B is not in the critical set since moving it would not affect the length of the critical path.

If not all instructions in a critical set can be moved, none are moved. The allocation of resources is similar to that in local schedulers. Overlapping resource holes are found for all resources required by each instruction. However, the holes must be within the instruction’s execution range, and wedged insertion is not performed, since the goal is to avoid increases to the critical path length of the destination block. An algorithm for performing global code motion in this manner is given in Figure 4.

Consider the problem of moving the instructions from Block 2 to Block 1 in Figure 5(a). Assume that there are three functional units and four registers available. The first critical set consists of instructions M1 and M2. Instruction M2 can be inserted in the functional unit hole following F and the register hole following D since the holes overlap. M2’s value must be spilled since the register hole is not available to the end of the DAG. M1 can be inserted in the functional unit hole following B and the register hole following G, which results from killing F. The value computed by M1 need not be spilled. The resulting DAGs are shown in Figure 5(b). Next M3 and M4 are moved up. Since M4 is selected to kill both M1 and M2 it can use the same functional unit and register as M2. Instruction M3 can use M1’s functional unit, and it will also take M1’s register, forcing M1 to use B’s register. B’s value must now be spilled around the inserted instructions and M3’s value is spilled before B’s value is reloaded. The resulting DAGs are shown in Figure 5(c).

Traditional global schedulers, based on list scheduling, are able to identify functional unit holes. However, since the scheduler is separate from the register allocator, it does not know if there are registers available for the instructions that it moves up. Similarly, these schedulers cannot recognize when instructions from other blocks should be moved
up above instructions in the block with slack time, since these schedulers usually schedule all instructions in the current block first. Resource Spacing can move instructions from other blocks above instructions with slack time in the current block when overlapping resource holes are available.

Several problems concerning the insertion of fill instructions must be considered. The code motion algorithms must determine if the critical path length of the source block is decreased when loads of moved values are inserted in it. The algorithms must also consider the impact of code duplication on the critical paths.

6 Experimentation and Concluding Remarks

We have implemented Resource Spacing using our experimental compiler tool, pdgcc. Pdgcc is a C compiler front-end which performs dataflow and dependence analysis and generates intermediate code in the form of PDGs. Both local reductions and global code motions have been implemented. The target architecture of the experiments is a VLIW architecture that supports speculative execution. Two configurations of different numbers of functional units and registers are used. In the target architecture all instructions execute in one cycle, except for memory access instructions which execute in two cycles. Although the framework supports all code motions for conditionals, the experiments are based on code motions for speculative execution of instructions.

The procedures used consist of six routines from the C version of the linpack benchmark. Execution profile information was collected for each region. The resulting execution times, after local and global Resource Spacing has been performed, are determined by multiplying the execution times of each region by the number of cycles required to execute the region.
The results for two target architectures are shown in Table 1. The columns labeled local give the speedup over a 1 wide VLIW architecture when only local scheduling using reductions are performed. The columns labeled global give the speedups when both local and global scheduling are performed. The size of the critical path reductions ranged from 1 to 6 cycles and averaged 2.6. The number of instructions moved ranged from 1 to 8 and also averaged 2.6.

The numbers show that Resource Spackling is able to exploit the parallelism available in the benchmarks within the constraints of the architecture's resources. Further, in all but two cases, ddot and idamax, global code motion using Resource Spackling made additional improvements. Upon examining the routines and their profile information, it was discovered that instructions were moved during global code motion, but that neither their source or destination regions were executed during the profiling runs. The routines ddot, idamax, and epsilon did not show any improvement when run on the 8 wide architecture because the resource requirement measurements showed that the 4 wide architecture provided sufficient resources.

Resource Spackling is a framework which is general enough to allow common architectural features to be incorporated. Separate ReuseR DAGs can be built for each type of resource provide, e.g., integer and floating point register files, and integer functional units and separate floating point adders and multipliers. Pipelines, resource demands that can be satisfied by several types of resources, and implicit resource demands also can be modeled in Resource Spackling[BGS94].

References


