Presented at IFIP WG 10.3(Concurrent Systems) Working Conference on Architectures and Compliation Techniques for Fine and Medium Grain Parallelism, Orlando, Fl., January 1993

URSA: A Unified ReSource Allocator for Registers and Functional Units in VLIW Architectures †

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

Department of Computer Science, University of Pittsburgh Pittsburgh, PA 15260

Keyword Codes: C.1.1; D.3.4
Keywords: Single Data Stream Architectures; Programming Languages, Processor

Abstract

The division of instruction scheduling and register allocation and assignment into separate phases can adversely affect the performance of these tasks and thus the quality of the code generated for load/store fine grained parallel architectures. Improved performance in one phase can deteriorate the performance of the other phase, possibly resulting in poorer overall performance. In this paper we present an approach that partitions instruction scheduling and register allocation and assignment into a new set of phases in an attempt to construct phases with minimal interaction. This approach uses a technique that unifies the problems of allocating registers and functional units. The technique, which consists of three phases, operates on a dependence DAG representation of the program. The first phase carries out the measurement of resource requirements and identifies regions with excess requirements. The second phase applies transformations that reduce the requirements to levels supported by the target machine. The final phase carries out the assignment of resources.

1. Introduction

In an effort to reduce the complexity of compilers, the process of translating and improving program code is traditionally split into multiple phases. In compilers for load/store fine grained parallel architectures this approach has led to separate phases for register allocation and instruction scheduling. This division of work does not consider the impact that the solution to one problem has on the other problem. The goal of register allocation is to minimize the number of accesses made to main memory. Thus, techniques have been developed that attempt to retain a limited number of values in registers as long as they are needed. The goal of instruction scheduling is to maintain a high utilization of the functional units. The concurrent execution of many instructions typically requires access to a comparable number of registers for input values and results. If register allocation is performed before instruction scheduling, additional dependences due to the reuse of registers are introduced, further restricting the scheduler. On the other hand, if instruction scheduling is performed before register allocation then any spill code that is introduced must be incorporated into the existing schedule. Thus, a good solution to one problem may prevent a good solution to the other and, more importantly, it may degrade the quality of the code generated.

---

† Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-9109089 to the University of Pittsburgh.

In this work we present a new technique, Unified ReSource Allocation (URSA), for VLIW architectures, which uses a unified approach and common representation for performing both register allocation and instruction scheduling. Furthermore, this technique defines a new set of phases, in which allocation of all resources is performed in one phase and assignment of all resources in a second. This grouping and ordering of the tasks removes the impact of dependences due to register reuse on instruction scheduling, while still being able to consider the register constraints prior to instruction scheduling. The unified approach treats allocation, or scheduling, of both resources as a single problem. The technique allows the effects of allocation options on the resources to be determined. For example, the technique can compare the effect that spilling of a value and scheduling the instructions in a particular order has on register demands. The allocation option that has the best overall effect can then be selected. The result is a integrated allocation of both resources that attempts to minimize the negative impact of the interactions on the quality of code generated.

The URSA technique consists of three major components: a representation of the program and the requirements for each resource, algorithms to measure the resource requirements, and transformations that reduce the resource requirements to levels supported by the target architecture. The unified approach uses a dependence DAG to measure the resource requirements and represent the program as it is being transformed. The use of a DAG allows all semantically correct schedules to be considered. Measures of resource requirements are found by determining the maximum number of each resource needed for any schedule. These measures are then used to identify the regions of the DAG that require reductions in resource requirements to meet the levels supported by the target machine. The resource requirement reduction transformations modify the DAG by adding sequence edges which remove schedules with excess resource requirements from consideration. Load and store instructions may also be added to spill registers. The relative benefit of several possible transformations can be compared by determining the amount that each transformation reduces the resource requirements and its impact on the critical path through the DAG.

Several techniques have been proposed to address the issue of interaction between the two problems for pipelined architectures [BEH91, GoH88, SwB90]. However all of these techniques still attempt to solve the problems separately while taking into account the interactions to a limited extent. These techniques attempt to maximize the number of resources used at each point in the program, within the limits of the target architecture, while having only a limited understanding of the impact on the remainder of the schedule. The underlying scheduler for all but one of these techniques is list scheduling. List scheduling has a restricted view of the impact of its decisions on the overall schedule. The priority function for each instruction, used to select which instructions to schedule next, is only an estimate of the impact on the remainder of the schedule. Furthermore, it cannot undo an earlier portion of the schedule if it discovers that its estimates were inaccurate. Our technique uses more information from a larger region of the program when performing allocation and scheduling. The effective goal of our technique is to maximize the average utilization of the resources by minimizing the execution time, without ever exceeding the limits of the target architecture. Goodman and Hsu [GoH88] also present a DAG driven technique. However, their technique uses prepass scheduling and does not have a mechanism for inserting spill code.

Section 2 gives an overview of the URSA technique. Section 3 presents the techniques for measuring the resource requirements. Section 4 presents the resource requirements reduction transformations and discusses their interaction and integration. Section 5 discusses the integrated application of URSA's transformations. Concluding remarks and future work are given in section 6.

2. Overview of the Unified ReSource Allocator (URSA)

The goal of both register allocation and instruction scheduling is to perform the allocation of resources to exploit instruction level parallelism. Instruction scheduling performs allocation and then assignment of different functional units to instructions that are to execute concurrently. Register allocation ensures that registers are available to store the values computed by instructions that are executed concurrently.

The purpose of the URSA is to modify the dependence DAG in such a way that its resource requirements cannot exceed the capacity of the target machine. Therefore, it is only concerned with the allocation of resources, and not their actual assignment. Using heuristics, URSA's goal is to approach the two NP-Complete problems of register allocation and precedence constrained scheduling in a unified manner to reduce their negative impact on each other. URSA uses a DAG that represents resource reuse information to measure a program's resource requirements and identify the regions that have excessive resource requirements for the application of the transformations. The applications of URSA's transformations are followed by resource assignment and code generation phases. The assignment phase is also responsible for handling any excessive requirements that were not identified by URSA's heuristics. The major steps of URSA are summarized in Figure 1.

The basic structure that is used for representing parallelism in straight line code is the dependence DAG. A DAG representation is suitable for exploiting parallelism present within basic blocks as well as parallelism across basic block boundaries [Fis81]. By constructing DAGs of traces, which are basic block sequences, trace scheduling allows code motion across basic blocks. Other techniques that also allow code motion across basic blocks include percolation scheduling and region scheduling [AiN88, GuS90]. However, they do not use an explicit DAG representation during code motion. In this work we use DAGs since they represent a partial order on the execution of instructions, which provides a basis for measuring resource requirements. Furthermore, a DAG allows the effects of program transformations on resource requirements to be directly assessed.

The edges in a DAG representation of a trace enforce the data dependences among the instructions and sequence the instructions to preclude illegal motion of code across branches. Thus, these edges preserve semantic correctness of the program. Additional edges can be added to introduce sequentiality, i.e. reduce parallelism, where needed to reduce resource requirements. For the purposes of the algorithms discussed in this work, the DAG representation is modified so that there is a single root node and a single leaf node. The single root and single leaf represent the entry to and exit from the basic block.

Details of the algorithms described in the subsequent sections can be found in [BGS92].

3. Measuring Resource Requirements

The measurements of register and functional unit requirements use the same algorithm and a common data structure, a Reuse DAG, which indicates which instructions can reuse a resource used by a previous instruction. The difference in the usage of the two resources is handled by different methods for constructing the Reuse DAG for each resource. Both the program DAG and the Reuse DAG contain two types of edges, data dependence edges and sequentialization edges added either by the trace scheduler or by URSA.

3.1. The Measurement Algorithm

Measuring the resource requirements consists of two tasks: determining from the DAG the maximum number of resources required and finding the regions in the DAG where the resource requirements exceed the available resources. The maximum requirements for a given resource, in a given region of the program, is the maximum amount of that resource required

```
Algorithm URSA( Trace )
{
  Construct the dependence DAG from Trace

  Measure the requirements for both functional units and registers:
    Construct the Reuse DAG for the resource
    Determine the number of resources required, given by the number
      of chains found using bipartite matching
    Use the chains to locate regions with excess resource requirements

  While there are regions with excess requirements do
  {  Reduce requirements by applying transformations to the DAG:
      Add sequential edges to reduce functional unit requirements
      Reduce register requirements by either
        Adding sequential dependence edges, or
        Inserting store and load instructions to spill values

    Update the measurements of resource requirements and regions
      with excess requirements
  }
  Assign registers and functional units

  Generate code
}
```

Figure 1 - Top level of URSA

under any schedule. It should be noted that a single schedule may not realize the maximum requirements for all resources. Furthermore, a single schedule may not be able to realize the maximum requirements for a given resource for all regions of a program. Thus the maximum resource requirements represent a worst case scenario.

The algorithm for obtaining the measurements of resource requirements operates on a DAG representing a partial order describing the dependences. A chain in a partial order is a subset of elements such that any pair of elements in the subset are related. Every path in a DAG is a chain in the corresponding partial order, but a chain is not necessarily a path since it may be noncontiguous. Figure 2(a) shows a basic block of code and Figure 2(b) shows the corresponding DAG. In Figure 2(b), the sets of nodes {A, B, F, K}, {C, E, I}, {D, G, J}, and {H} are all chains.

Definition 1 Let relation Q be a partial order on set S †. A *chain* is a set $S' \subseteq S$ such that if $a, b \in S'$ then either $(a, b) \in Q$ or $(b, a) \in Q$.

Definition 2 A *decomposition* of a partial order P is a partition of P into chains. A decomposition is *minimal* if there is no other decomposition with fewer chains.

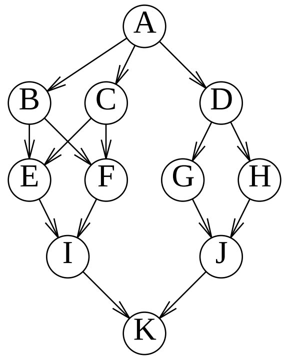
If two nodes are independent, then they may be executed in parallel. The following theorem relates the maximum amount of parallelism to a minimal chain decomposition.

---

† A relation Q on a set S is a *partial ordering* iff Q is reflexive, transitive, and anti-symmetric.

```
A: load v
B: w = v * 2
C: x = v * 3
D: y = v + 5
E: t1 = w + x
F: t2 = w * x
G: t3 = y * 2
H: t4 = y / 3
I: t5 = t1 / t2
J: t6 = t3 + t4
K: z = t5 + t6
```

(a)



(b)

Figure 2 - Example 3-address code and corresponding DAG

Theorem 1 The maximum number of independent elements in a partial order is equal to the number of chains in a minimal decomposition.

Proof: see [Dil50].

The DAG in Figure 2(b) can be minimally decomposed into a set of four chains, such as { {A, B, E, I, K}, {C, F}, {D, G, J}, {H} }. Thus, at most four nodes at a time can execute in parallel.

If the resources needed can be represented as a partial ordering on the instructions, the task of computing maximum resource requirements can be performed using Theorem 1. The partial ordering on the nodes of a DAG, with respect to resource R, is defined as follows:

Definition 3 Let $CanReuse_R$ be a relation on nodes of the DAG for resource type R indicating if a resource instance r of type R used by a node can be reused by one of its descendants, i.e., $(a, b) \in CanReuse_R$ iff there is a node c that ends a's use of r and $c \in Ancestors(b) \cup \{b\}$.

In other words, under the relation $CanReuse_R$ there is no schedule such that node b can execute while resource instance r is still in use as a result of executing a. The CanReuse relation varies from one resource to another. For example, the use of a functional unit is finished after the instruction has been executed. A register is still in use after the instruction that placed a value in the register completes execution.

Definition 4 A $Reuse_R$ DAG $(N, \bar{E})$ for resource type R is constructed from a program DAG (N, E). All edges (a, b) in $\bar{E}$ must meet the following two conditions:

1) $(a, b) \in CanReuse_R$, and

2) $\not\exists c \ni (a, c) \in CanReuse_R \text{ and } (c, b) \in Q$.

The second condition simply eliminates transitive edges from the $Reuse_R$ DAG. Although this condition is not necessary, it simplifies later discussions and techniques. $Reuse_R$ DAGs for functional units and registers are denoted as $Reuse_{FU}$ and $Reuse_{Reg}$ respectively. The notation $Reuse_R$ is used when reference is not restricted to a particular resource. The DAG in Figure 2(b) is both a program DAG and a $Reuse_{FU}$ DAG.

Definition 5 An *allocation chain* for resource R is a chain $n_1, n_2, \cdots n_l$ such that $(n_i, n_{i+1}) \in Reuse_R$ for any consecutive members n_i, n_{i+1} in the chain.

The regions of the program that require the application of URSA's transformations are those that contain sets of instructions that if executed concurrently would require more resources than are available. URSA's transformations do not require enumeration of all excessive sets, but only the sets of allocation subchains that are independent of each other and whose size exceeds the number of resources available.

Definition 6 An *Excessive chain set* $ES_R = \{EC_1, EC_2, \cdots EC_m\}$ for resource type R is a set of allocation subchains EC_i from the minimum decomposition such that

1) There are excessive requirements, i.e., $m > |R|$.
2) All nodes exist in at least one set I of m independent nodes: $\forall_{n_i \in EC_i} \exists I$ such that $n_i \in I$, $|I| = m$, and $\forall n_j, n_k \in I$ n_j, n_k are independent.
3) There are no edges from one chain head to another and there are no edges from one chain tail to another†, i.e., $\forall_{1 \le i,j \le m}$ EC_i's head and EC_j's head are independent, and EC_i's tail and EC_j's tail are independent.

A $Reuse_R$ DAG can be decomposed into allocation chains. If there are sufficient resources, each allocation chain can be assigned a different resource. However, if there are insufficient resources, these chains provide a measure of the resource requirements. Clearly, all chains in a $Reuse_R$ DAG are allocation chains. Therefore, by Theorem 1, a minimum decomposition of a $Reuse_R$ DAG into allocation chains gives the maximum resource requirements of R for the original DAG.

Ford and Fulkerson [FoF65] have shown that the problem of finding a minimum chain decomposition can be solved by transforming it into a maximum bipartite graph matching problem. The bipartite graph represents all possible pairs of nodes $(a, b) \in CanReuse_R$. Since each node in the $Reuse_R$ DAG must participate in exactly one chain, a maximum matching finds a minimum number of allocation chains.

The second step in resource requirements measurement is to identify all of the excessive chain sets. Each set is contained in a local region such that no instructions outside of the region need to be considered when applying transformations to remove excessive requirements. Formally, this local region is a hammock. Note that since the original DAG is modified to have only one root and one leaf node, it is a hammock.

The computation of the excessive chain sets requires that the projection of the minimum decomposition for the DAG onto any hammock also be a minimum decomposition. The decomposition algorithm in [FoF65] only guarantees minimum decomposition for the entire DAG, but not for all hammocks nested within the DAG.

A bipartite matching algorithm based on augmenting paths can be modified to find matchings that use only edges that will result in a decomposition that is minimal for all nested hammocks. The modification consists of using the difference in hammock nesting level between the source and sink nodes to prioritize the edges selected for augmenting paths. The prioritization is implemented by adding the edges to the bipartite graph in sets based on their

---

† The head of a chain is the node that has no predecessors in the chain. The tail of a chain is the node that has no successors in the chain.

priority. As each set is added, the normal augmenting path matching algorithm is called to find the matching. All edges that do not cross from one hammock to another have the highest priority, and thus the algorithm can first find a matching using only these edges. The algorithm then adds the sets of edges based on the difference in nesting level between the source and sink nodes of each edge. This modified algorithm has a worst case time complexity of $O(N^3)$.

Once the $Reuse_R$ DAG is properly decomposed, the excessive requirements sets can be found. All excessive chain sets can be found in a reasonably straightforward manner by examining contiguous allocation subchains and removing any heads and tails that are related to other heads or tails in the other subchains being considered. Consider the minimal decomposition { {A, B, E, I, K}, {C, F}, {D, G, J}, {H} } for the DAG in Figure 2(b). The tail of the third chain is **J**, which is dependent on the tail of the fourth chain, **H**, so **J** is removed from the excessive set. Likewise, **I** and **K** are removed. The heads of the second and third chains, **C** and **D**, are all dependent on the head of the first chain, **A**, so **A** is removed from the excessive set. Likewise, **D** is removed. The set of chains in the excessive set is { {B, E}, {C, F}, {G}, {H} }. The remaining dependences between nodes in different excess chains will be considered when the transformations are applied.

3.2. Measurement specifics for functional units and registers

The definition of $CanReuse_R$ differs for registers and functional units due to the difference in how they are used. A functional unit is only in use while an instruction is being executed; once the instruction has been executed the functional unit is available for reuse. In nonpipelined architectures if instruction **b** is dependent on instruction **a**, then **b** cannot begin execution until **a** has completed. Therefore, $CanReuse_{FU}$ is the partial order represented by the program dependence DAG, and the computation of the functional unit requirements and excess sets can be performed in polynomial time.

A register is used to hold a value from the time that the defining instruction executes until the value is killed by the last instruction that uses it. Therefore, the definition of $CanReuse_{Reg}$ requires that the killing instruction be identified for each value defined, i.e., the last use instruction to execute. However, URSA does not assume a specific schedule. Since the purpose of the resource requirements computations is to find the worst case scenario, the use instruction that would maximize the number of registers required is selected to be the killing instruction. Let $Kill(a)$ be the function that returns the node selected to kill node a's value. Then

$$CanReuse_{Reg} = \{(a, b) \mid b = Kill(a) \text{ or } b \in Descendents(Kill(a))\}$$

In many cases the definition of $Kill()$ is straightforward. However, there are several cases where defining $Kill()$ is NP-Complete. In these cases the values of a set of nodes can be alive at the same time as a number of their dependents. $Kill()$ must be defined to maximize the number of dependents that can be alive at the same time as their ancestors. This is accomplished by finding the minimum sized set of descendants that kill all of their ancestors.

Theorem 2 Defining $Kill()$ for all nodes in the DAG is NP-Complete.

Proof: By reduction to the Minimum cover problem [BGS92].

The sub-DAG {B, C, E, F} of the DAG in Figure 2(b) is an example of the difficult case. An optimal solution to the minimum cover problem for this sub-DAG will choose the same node to kill both **B** and **C**. Let the solution be **F**. Then, $Kill(B) = Kill(C) = F$, so $(B, F) \in CanReuse_{Reg}$, $(C, F) \in CanReuse_{Reg}$, $(B, E) \notin CanReuse_{Reg}$, $(C, E) \notin CanReuse_{Reg}$. Thus, three allocation chains are required to decompose this sub-DAG.

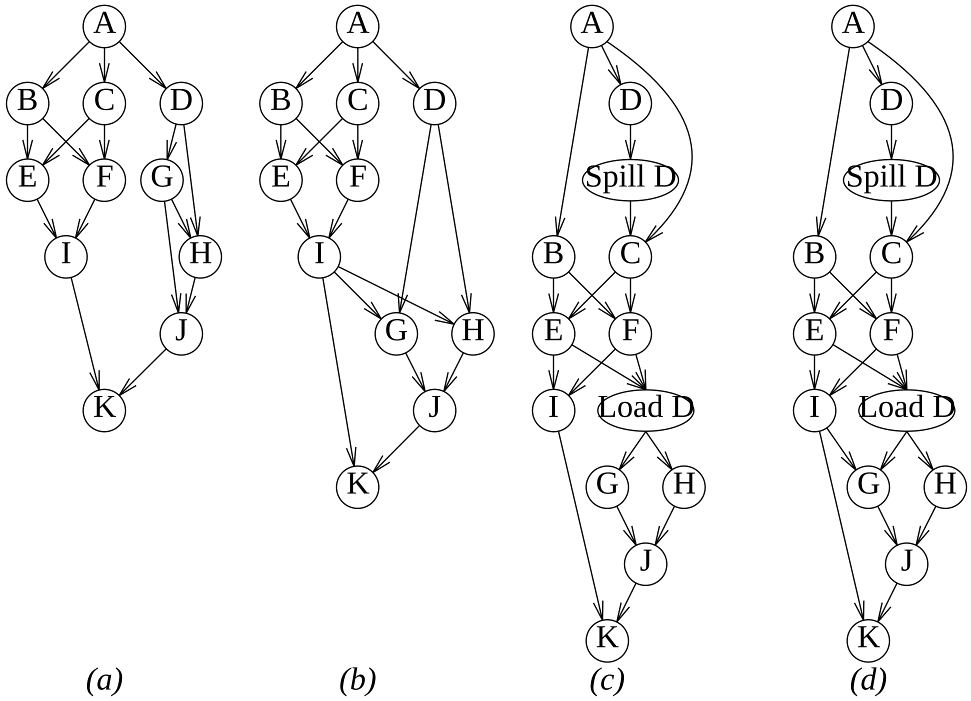


Figure 3 - Transformations on example DAG

4. Reducing Resource Requirements

Once the excessive chain sets are identified, transformations must be applied to remove the excess requirements. Each resource has different properities that require different transformations. As an example, consider the DAG in Figure 2(b). It requires five registers and four functional units to exploit all available parallelism. There is one transformation for reducing functional unit requirements: adding dependence edges to sequentialize independent nodes in an excessive chain set. In Figure 3(a) an edge has been added from **G** to **H**, reducing the functional unit requirements to three.

There are two types of transformations that can be used to reduce register requirements. The first is a sequentializing transformation. Sequentialization of register uses is complicated by the values required by the instructions on the sinks of the sequentializing edges. Such values which are alive during the execution of instructions that are not delayed contribute to the resource requirements. If nodes **G** and **H** are delayed until after the execution of **I**, as shown in Figure 3(b), the value generated by **D** is alive and requires a register during the execution of nodes **B**, **C**, **E**, **F**, and **I**. Thus the register requirements are reduced to four. Because of values such as the one generated by **D**, sequencing register requirements is not always possible. In this case a transformation that introduces spill code to sequentialize register requirements must be used. In Figure 3(c) the value generated by **D** is spilled and not reloaded until a register is available for it, reducing the register requirements to three. Figure 3(d) shows the DAG after a combination of transformations has been applied to reduce the functional unit requirements to two and the register requirements to three.

4.1. Reducing Functional Unit Requirements

The only way to remove excess functional unit requirements, i.e., instruction parallelism, is to add sequentiality between independent instructions in the excessive chain set. This is accomplished by adding sequential dependence edges to the DAG representation of the program. The goal, while adding the edges to remove the excess functional unit requirements, is to minimize the overall execution time of the program. Let X be the amount of excess functional unit requirements that must be removed from the excess set and let the sets of nodes for sources and sinks for the sequential dependences edges to be added be S and T respectively. Consider if the i^{th} edge is added from the node in S that is i^{th} closest to hammock's entry to the node in T that is i^{th} closest to the hammock's exit. The last edge added will be the longest path from the entry to the exit using one of the added edges because it uses the node in S and the node in T that are furthest from their respective ends of the hammock. A better approach is to average the length of the resulting paths from the entry to the exit node. This is accomplished by adding an edge from the tail of the chain in the excessive set whose tail is i^{th} closest to the hammock's entry node to the head of the chain in the excessive set that is i^{th} closest the hammock's entry node. We call this *ideal sequence matching*.

Assume that the DAG in Figure 2(b) must have its functional unit requirements reduced from four to three. Then the sets $S = G$ and $T = H$ meet the conditions required and an sequential dependence edge is added from **G** to **H**, as shown in Figure 3(a).

It can be shown that if S consists of the X nodes closest to the H's entry node and T consists of the X nodes closest to H's exit node, then the transformation is optimal with respect to the execution time of the program. Since precedence constrained scheduling is NP-Complete and locating all excessive chain sets is polynomial, the computation of optimal sets S and T is NP-Complete. Therefore, the heuristic for finding S and T attempts to choose nodes as close as possible to the entry and exit nodes respectively. This heuristic begins with sets S and T containing the X nodes closest to the respective hammock end points. It then tests the conditions by trying to apply the transformation. If the application fails it replaces a node in T with one closer to the entry node and repeats the test. Note that any subset of either the heads or tails of the excessive subchains will satisfy the conditions. The time required to perform the test is $O(Nm)$, where N is the number of nodes in the hammock containing the excessive set, and m is the number of allocation chains allowable, i.e., the number of resources available. The number of times the test must be performed is N, giving the heuristic an overall worst case time complexity of $O(N^2m)$.

There are cases when the transformation must be applied several times within the same hammock. These occur when one application of the transformation cannot remove all of the excess requirements, either because the heuristic failed to find optimal sets, or when the measured requirements are more than twice the amount supported. In such cases the portions of the chains below each node in T and above each node in S are removed from the chains in the excessive set, and the transformation is applied again.

4.2. Sequentializing Register Requirements

From an allocation point of view, the primary difference between functional units and registers is that a functional unit is in use only while an instruction is executing, while a single use of a register spans the execution of multiple instructions. The effects of this difference appears when computing the revised requirements after sequential dependence edges are added with the goal of reducing register requirements.

Definition 7 A sub-DAG A is said to be *nonsupporting* of a sub-DAG B if there are no edges from any node in A to any node in B in the DAG containing A and B.

Definition 8 Let H be a hammock with two sub-DAGs SD_1 and SD_2 where SD_2 is nonsupportive of SD_1, both sub-DAGs consist only of nodes from an excessive chain set, and SD_2 is to be sequenced after SD_1. Then the transformed hammock is divided into stages $Stage\,1$ and $Stage\,2$, given by

$$Stage\,1 = \bigcup_{r \text{ is a root of } SD_2} Ancestors(r)$$

$$Stage\,2 = \bigcup_{r \text{ is a root of } SD_2} Descendants(r) \cup r$$

As an example, let SD_1 = {B, C, E, F, I} and SD_2 = {G, H} for the DAG in Figure 2(b). Then *Stage1* = { A, B, C, D, E, F, I} and *Stage2* = {G, H, J, K}.

The resource requirements for the transformed hammock are given by *Max(Chains(Stage1), Chains(Stage2))*, where *Chains(Set)* returns the set of chains from the minimum decomposition that cover *Set*.

To be useful, a register sequentializing transformation must reduce at least one of *Chains(Stage1)* and *Chains(Stage2)* to require no more than the number of registers available. In addition, the transformation should have a minimal effect on the overall critical path. As in the case of sequencing functional unit requirements, a single application of the transformation may not be able to remove all excess requirements. Such a situation may also occur when there is a sub-DAG within the appropriate stage that is nonsupportive of SD_1 and SD_2.

For some $Reuse_{Reg}$ DAGs, no sets S and T can be found, either because there are dependences from nodes in T to nodes in S, or because the resulting transformation would not reduce the registers requirements in SD_1. The heuristic for finding S and T operates in the same manner as the functional unit heuristic, and thus has a worst case time complexity of $O(N^2m)$.

Because a live value extends forward from its definition point to its uses, better results are expected when $Stage\,1$ rather than $Stage\,2$ is required to have no more than $|Regs|$ chains. The best $Stage\,2$ will be the one that has the fewest chains and minimizes the critical path of the overall DAG.

As an example, consider the DAG in Figure 2(b), which requires five registers because the values from nodes **B**, **C**, **E**, **G**, and **H** can all be alive at the same time. If S = {I} and T = {G, H} then *Stage1* = {A, B, C, D, E, F, I} and *Stage2* = {G, H, J, K}. Further, *Stage2* is nonsupportive of *Stage1*, *Chains(Stage1)* = 4 and *Chains(Stage2)* = 3. Thus S and T satisfy all of the conditions of the transformation. Adding sequential dependence edges from **I** to **G** and **H**, as shown in Figure 3(b), reduces the register requirements from five to four.

4.3. Using Spills to Reduce Register Requirements

Spilling registers is similar to sequencing registers, in that register requirements are delayed until there are sufficient registers available. However, the application of spilling transformations does not need to consider the impact of nonsupporting sub-DAGs on the resource requirements of the first stage. Effectively, the transformation is the same except that the values computed by the roots of the nonsupporting DAG SD_2 are spilled before SD_1 can start, and are reloaded after SD_1 finishes execution.

The spill introducing transformation finds sub-DAGs SD_1 and SD_2 in the same manner, except that there must only be sufficient resources for SD_1, and not necessarily for all of *Stage1*. The roots of SD_2 are computed and their values are spilled prior to SD_1's roots. The reloads of the values are placed after SD_1's leaves.

Because of the relaxed requirements on the sub-DAGs SD_1 and SD_2, the sets S and T can always be found and application of the transformation will always sufficiently reduce the register requirements in a segment of the program. The heuristic for finding S and T operates in the same manner as the functional unit heuristic, and thus has a worst case time complexity of $O(N^2 m)$.

Consider the sub-DAGs SD_1 = {B, C, E, F} and SD_2 = {D} of the DAG in Figure 2(b). Figure 3(c) shows the transformed DAG. The value of node **D** is computed and spilled before **B** or **C** are executed. **D**'s spilled value is reloaded after **E** and **F** execute, and then **G** and **H** can execute.

5. Application of the Transformations

All three transformations presented operate on the same DAG, allowing them to be applied in any order or in an integrated manner. Since all transformations sequentialize instructions, an application that reduces the requirements of one type of resource may also reduce the requirements of another resource. It should be noted that the techniques can be used when there are several classes of a resource, such as integer and floating point registers. In such a case a separate $Reuse_{R_C}$ DAG is constructed for each class C of resource R.

A possible heuristic for minimizing the critical path of the transformed DAG is to select the transformation that best reduces the excess requirements of all resources. Each transformation is tentatively applied, and the resource requirements of the transformed DAG are measured for all resources that had excess requirements. The transformation that is best with respect to the combination of minimizing the critical path and reduction of all excess requirements is selected and applied. The process is repeated while there are excess requirements. Thus, the application of both register and functional unit transformations can be integrated.

If multiple classes of resources are not present then the transformations can be applied in phases, and the interactions between the transformations can be examined to determine their ordering. The effect of the sequentialization transformations for both functional units and registers is to reduce the width of the DAG. Neither transformation can increase the requirements of either resource. The application of register sequentialization is also likely to reduce functional unit requirements, whether they are excessive or not. The application of functional unit sequentialization can reduce register requirements, but will force long lifetimes for some of the values. These interactions suggest that register sequentialization has a more significant impact on the reduction of functional unit requirements than functional unit sequentialization has on register requirements.

Consider target architectures that require a functional unit to perform loads and stores. An introduced spill instruction can execute concurrently with the same set of instructions as the value producing instruction, and so can use the same functional unit. On the other hand, an introduced load may require an additional functional unit if it can execute concurrently with other parents of its dependents. The load cannot share a dependent's functional unit without impact if there are at least as many such instructions as there are dependents of the load instruction. Therefore, introducing a spill cannot increase functional unit requirements, while introducing loads will increase functional unit requirements in some cases.

The above discussion suggests that both register transformations should be applied in a single phase, and that the functional unit transformations can be applied in a subsequent phase. The register phase should consider both register transformations for excessive register sets. If both transformations can be applied to reduce the same excessive set, then the one that minimizes the critical path through the DAG should be selected. If both have the same impact on the critical path, then the register sequencing transformation should be applied, since it does not require the use of additional resources to access main memory.

6. Summary and Future work

This paper has presented a unified technique that integrates register allocation and instruction scheduling for VLIW architectures. This technique differs from other integration techniques in that it performs resource assignment after all resources are allocated. Thus, most of the interaction between the two allocation problems is removed. The technique consists of an algorithm to measure the resource requirements in each region of the program and transformations that can be applied in an integrated manner to reduce the resource requirements while attempting to minimize the impact on the execution time of the program.

This technique is currently being implemented in C on a Sun workstation. It uses an existing C compiler front end that first constructs a Program Dependence Graph (PDG) and then generates the initial dependence DAG for each trace.

Several extensions to this work are currently being investigated. Heuristics to integrate the application of transformations when there are multiple classes of resources are being developed. The techniques are being combined with loop unrolling to create a new resource constrained software pipelining technique. Extensions to handle the problems caused by interlocks in pipelines are also being developed, so that superscalar architectures can be targeted. Lastly, the resource requirements measurements will be used to determine when and which parallelizing transformations would be beneficial.

References

[AiN88] A. Aiken and A. Nicolau, A Development Environment for Horizontal Microcode, *IEEE Trans. on Software Engineering 14*, 5 (May 1988), 584-594.

[BGS92] D. A. Berson, R. Gupta and M. L. Soffa, URSA: A Unified ReSource Allocator for Registers and Functional Units in VLIW Architectures, Technical Report 92-21, University of Pittsburgh, Computer Science Department, Nov. 1992.

[BEH91] D. G. Bradlee, S. J. Eggers and R. R. Henry, Integrating Register Allocation and Instruction Scheduling for RISCs, *Proc. Fourth International Conf. on ASPLOS*, Santa Clara, CA, April 8-11, 1991, 122-131.

[Dil50] R. P. Dilworth, A Decomposition Theorem for Partially Ordered Sets, *Annuals of Mathematics 51*(1950), 161-166.

[Fis81] J. A. Fisher, Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Trans. on Computers C-30*, 7 (July 1981), 478-490.

[FoF65] L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, N.J., 1965.

[GoH88] J. R. Goodman and W. Hsu, Code Scheduling and Register Allocation in Large Basic Blocks, *Proc. of the ACM Supercomputing Conference*, 1988, 442-452.

[GuS90] R. Gupta and M. L. Soffa, Region Scheduling: An Approach for Detecting and Redistributing Parallelism, *IEEE Trans. on Software Engineering 16*, 4 (April 1990), 421-431.

[SwB90] P. Sweany and S. Beaty, Post-Compaction Register Assignment in a Retargetable Compiler, *Proc. of the 23rd Annual Workshop on Microprogramming and Microarchitecture*, Nov. 1990, 107-116.