GURRR: A Global Unified Resource Requirements Representation*

David A. Berson berson@cs.pitt.edu
Rajiv Gupta gupta@cs.pitt.edu
Mary Lou Soffa soffa@cs.pitt.edu

Computer Science Department
University of Pittsburgh
Pittsburgh, Pa. 15260
Fax: (412) 624-5249

Abstract

When compiling for instruction level parallelism (ILP), the integration of the optimization phases can lead to an improvement in the quality of code generated. However, since several different representations of a program are used in the various phases, only a partial integration has been achieved to date. We present a program representation that combines resource requirements and availability information with control and data dependence information. The representation enables the integration of several optimizing phases, including transformations, register allocation, and instruction scheduling. The basis of this integration is the simultaneous allocation of different types of resources. We define the representation and show how it is constructed. We then formulate several optimization phases to use the representation to achieve better integration.

1 Introduction

Recent research has proposed several methods to integrate back-end phases of instruction level parallelism (ILP) compilers in an effort to take into account the interactions among the phases and improve the quality of code generated. However, the primary phases of interest, register allocation and instruction scheduling, use different representations, thus restricting the degree of integration. As a result, little work has examined complex interactions such as performing code transformations with the foreknowledge of their impact on register allocation.

In this paper we present a representation, GURRR, that augments the control and data dependence information of a Program Dependence Graph (PDG) with resource requirements and availability to enable better integration of allocating registers and scheduling instructions (allocating functional units). The allocation of registers and functional units is made while considering the impact of the allocation decisions on the overall execution time of the program. We show how this representation can also be used by other phases to determine their impact on register allocation and instruction scheduling and thus, their impact on the overall execution time of the program.

Register allocation is traditionally performed by creating an interference graph to represent conflicts between values that can be alive simultaneously. The nodes in the graph are then colored with colors indicating the register assigned to a node's live value. Recent techniques have been developed to compute the additional interferences that can occur due to parallel instruction execution and instruction reordering [Pin93, NP93]. However, the interference graph does not provide precise information to indicate the effect of an allocation decision on the execution time of the program.

Instruction scheduling is typically performed on a dependence DAG using some variant of list scheduling. While the dependence DAG can be used to compute the live range interferences, it is not appropriate for making register allocation decisions. The DAG represents a collection of semantically correct schedules rather than a single schedule, and so it does not precisely indicate when each live range will begin and end.

Instruction scheduling and register allocation have a fundamental interaction related to the order in which they are performed. If instruction scheduling is performed first it may produce a schedule that will require values in registers to be spilled. The instructions for performing the spilling must be inserted into the schedule, usually requiring a second scheduling phase. If register allocation is performed first it can reduce the amount of parallelism available for scheduling due to the introduction of false dependencies on registers. Recent works have developed methods to incorporate some information from one phase into the other, but are unable to fully represent register allocation and instruction scheduling in a single representation and thus do not achieve the full integration of allocating functional units and registers simultaneously [GH88, NP93].

Little work has been done on integrating other back-end phases as well. Typically other back-end phases, such as global code motion, code optimizations, and loop transformations are performed without considering their impact on register allocation. In addition, little consideration is given to selecting transformations that expose a level of parallelism appropriate for the architecture. Instead, transformations are applied to expose as much parallelism as possible,

*Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-91090809 to the University of Pittsburgh

which is then scheduled. The result is that too much parallelism may be exposed and then must be removed by the scheduler. Region Scheduling does attempt to expose an appropriate amount of parallelism, but uses coarse estimates that do not consider register demands [GS90]. The special case of exposing parallelism through software pipelining has received attention for scheduling the resulting parallelism, [Lam88, AN88b] and more recent work has also considered register allocation [RLTS92, NG93].

To address the interactions between register allocation and instruction scheduling phases, we present a new representation that incorporates resource requirements and availability. This representation permits the impact of each decision in a phase to be determined from several key indicators of program execution. The most important indicator is an estimate of execution time. The execution time is directly affected by the capability to effectively allocate the architecture's resources, e.g., registers and functional units. To enable effective resource allocation without increasing the execution time, our representation indicates how many of each resource type are required by the code segment of interest, as well as the number of each resource type available at that location. In addition, the integration of transformations with resource allocation requires the representation to be incrementally updateable as transformations are applied and resources are allocated.

We define GURRR in section 2 and describe its construction in section 3. Section 4 demonstrates how several phases of an ILP compiler can be integrated by formulating them in terms of GURRR. In section 5 we show how GURRR can be incrementally updated. Section 6 compares GURRR to related intermediate representations. Finally, we summarize the work in section 7.

2 Integrated Resource Allocation Representation

2.1 Integrated Resource Allocation Properties

The goal of this work is to develop an intermediate representation that allows the compiler back-end to fully integrate phases that allocate and schedule different types of resources, such as registers and functional units. A single representation enables such integration by indicating where and how all resources of interest are used and available, as well as information on factors that affect the execution time of the program, such as critical path lengths and execution counts of regions. We refer to the capability to allocate multiple types of resources simultaneously as *unified resource allocation.*

Algorithms that integrate resource allocation need resource usage information to make effective allocation decisions which have a minimal impact on the execution time of the program. Resource allocation decisions only need to be made when there are locations in a program segment that require more instances of a resource than are available. Advanced resource allocation algorithms, such as those based on the *Measure and Reduce* paradigm [BGS93] and to some extent global schedulers, move instructions from locations where there are insufficient instances of a resource to locations where extra resource instances are available. Thus, the resource usage information must indicate all locations where resources are either over-utilized or under-utilized. Since the architectures targeted in this work exploit ILP, a representation must take into account the ability to schedule instructions in parallel.

We identify the following set of properties that an intermediate representation should satisfy to support unified resource allocation. We refer to these properties as the *Unified Representation Properties.*

Property 1 (Unified Representation) *The representation can be used to determine the impact of a resource allocation or set of resource allocations on all resource demands in all segments and the execution time of the program.*

Property 2 (Measurability) *The representation enables measurement of all segments' demands for all resources. A resource measurement is* **precise** *if it indicates the minimum number of copies of the resource needed to exploit all parallelism uncovered in each program segment.*

Property 3 (Resource Usage) *The representation identifies all locations in each segment where resources are either over-utilized or under-utilized.*

Property 4 (Executability) *The representation indicates if each program segment in its current state can be executed using the available resources. A program segment is* **executable** *if and only if for each resource the number of copies required is less than or equal to the number of copies available. A program is executable if and only if all segments are executable.*

Ideally, the representation should supply precise resource measurements. In previous work we have shown that computing precise register measurements for parallel architectures or when code reordering is considered is NP-complete [BGS93]. Thus, there is a trade-off between the precision of the measurements and the time taken to compute them. We have developed fast heuristics for measuring register requirements that are demonstrably precise.

The measurability of the representation allows resource usage information for all resources to be computed. An intermediate representation that provides resource usage information for all resources enables unified resource allocation.

2.2 GURRR

GURRR is an intermediate representation that meets the unified representation properties and is used to investigate unified resource allocation algorithms. To support a variety of parallelization techniques, including powerful code motions, GURRR is based on a modified form of the Program Dependence Graph (PDG).

We introduce the Instruction Program Dependence Graph to represent instruction level parallelism not explicitly expressed in the traditional statement level PDG. Since a single program statement may result in several intermediate code statements, representing the program at the intermediate code level permits access to more ILP. To support a wider range of code motions we convert the representation to Static Single Assignment form (SSA). Special instruction nodes can be added to carry loop and array access information to enable the exploitation of medium grain parallelism as well. Compilers using PDG based representations perform resource allocations on a region by region basis. Thus, regions correspond to the program segments mentioned in the unified representation properties.

The *Instruction PDG* (IPDG) is a graph $G = (N, E)$, in which the set of nodes, N, is a union of the following node types.

1. Instruction nodes, $\mathcal{I}$, are similar to statement nodes found in traditional PDGs, but represent intermediate opcodes.
2. Region nodes, $\mathcal{R}$, in the PDG identify a unique set of execution conditions or control dependencies.

The set of edges, E, is a union of the following edges types.

1. Control dependence edges, $\mathcal{C} \subset \{\mathcal{I} \times \mathcal{R}\} \cup \{\mathcal{R} \times \mathcal{I}\}$, connect the region node to the instruction and subregion nodes that execute under the conditions that it identifies. Control edges are also added from the instruction nodes specifying those conditions to the region node.
2. Data dependence edges, $\mathcal{D} \subset \{\mathcal{I} \cup \mathcal{R}\} \times \{\mathcal{I} \cup \mathcal{R}\}$, connect the instruction nodes and represent the dependence of the instruction nodes on data values computed by earlier instruction nodes. In addition, data dependence edges are added from the instruction nodes defining values to the region nodes containing uses of the values to summarize the dependence of the region as a whole on data values computed by earlier instructions.
3. Transitive data dependence edges, $\mathcal{D}_T \subset \{\mathcal{I} \cup \mathcal{R}\} \times \{\mathcal{I} \cup \mathcal{R}\}$, indicate indirect dependencies between nodes due to a sequence of data dependencies. The addition of these edges simplifies the computation of the ordering of nodes within a region.

GURRR extends the IPDG to include the resource usage information required to meet the unified representation properties. In addition to summarizing control dependence information, region nodes in GURRR are used to summarize resource usage information. Since the regions are organized in a hierarchical manner, the summary for a region must include resource usage information for both the instructions and subregions that it contains. Regions in GURRR also store execution counts, indicating how many times the region is expected to be executed during a run of the program. This information enables the allocation algorithms to make better decisions on how many resources to allocate in each region.

The Measure and Reduce allocation scheme used in our work explicitly decides which instructions should be delayed until all resources that it requires are available. To support these scheduling decisions GURRR must represent additional constraints placed on the ordering of nodes. GURRR must also contain information used to measure the resource requirements. A part of this information is identifying which instructions can share an instance of a resource. The following additions are made to the IPDG's nodes and edges to meet these requirements and obtain GURRR.

1. Resource hole nodes, $\mathcal{H}$, represent locations in the program where resources are under-utilized. Each hole node is annotated with the resource availability characteristics.
2. Temporal dependence edges, $\mathcal{T} \subset N \times N$, are used to represent sequential dependencies. These edges are used to supply additional ordering constraints on the nodes, such as placement of hole nodes, and instruction and region node scheduling.
3. Reuse edges, $\mathcal{U} \subset \{\mathcal{I} \cup \mathcal{R}\} \times \{\mathcal{I} \cup \mathcal{R}\}$, connect nodes that can temporally share an instance of a resource under any schedule allowed by all of the dependencies in a region. A separate set of reuse edges is used for each resource type.

The stipulation that Reuse edges are added only when a resource can be shared under all semantically correct schedules allows for parallel execution and code reordering. An allocation algorithm, such as Measure and Reduce, can select any allowable schedule and determine the worst case resource requirements. When only a single schedule, such as the original order of the sequential source code, is used the resource requirements measurements are less precise, since fewer overlaps of uses of resources are accounted for.

Reuse edges are the basis of the resource requirements measurement algorithm. The measurement algorithm uses these edges to find as many instructions as possible that can share a single copy of a resource. The resource requirements measurements for a region consist of two pieces of information: *allocation chains* and *excessive sets.*

Definition 1 *An* **allocation chain** *for resource R is a set of nodes that are fully ordered by the reuse edges for resource R. Thus, by definition all nodes in an allocation chain can temporally share one instance of resource R.*

Definition 2 *An* **excessive set** *for resource R is a set of nodes that can execute in parallel and together require more instances of resource R than are available.*

Proper computation of allocation chains indicates the total number of instances of a resource required. Examination of the allocation chains allows all locations where a resource is either over-utilized or under-utilized to be identified. Excessive sets are the nodes in the locations where a resource is over-utilized.

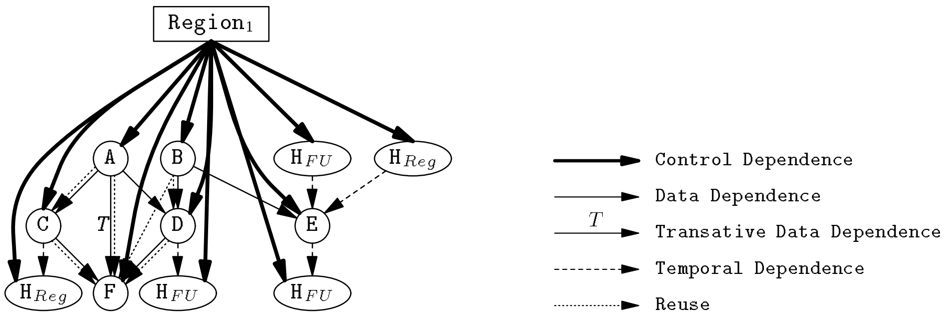
The resource requirements of a region are measured by finding the minimum number of disjoint allocation chains that contain all instruction and subregion nodes in a region [BGS93]. The reuse edges composing allocation chains are marked as such. The allocation chains in a region are used to locate the two indicators of resource usage levels: excessive sets and resource holes. Excessive sets are annotated in the region node. Resource hole nodes are placed on the allocation chains to indicate the location and characteristics of the resources holes.

Figures 1(a) and 1(b) show a simple program and the corresponding GURRR. The target architecture has three functional units and three registers. Control, data, and temporal dependencies, and reuse edges are indicated by bold, normal, dashed, and dotted lines, respectively. To improve readability only the reuse edges for registers have been drawn. The selection of the nodes that can reuse the registers used by instructions `A` and `B` is described elsewhere [BGS93].

To be considered useful for unified resource allocation GURRR must satisfy the unified representation properties. GURRR satisfies the Measurability property by using the reuse edges and allocation chains to compute each region's requirements for all resources. As discussed in section 3, the resource measurements are precise for functional units and usually precise for registers. GURRR provides resource usage information for all resources on the IPDG. All types of dependencies are represented by the various types of edges in GURRR, allowing the execution time of a region to be computed. The combination of all of this information on a region by region basis and in hierarchical summaries satisfies the Unified Representation property. In each region the excessive sets and resource hole nodes identify all locations that

```
1:  load A
2:  load B
3:  C = A - 10
4:  D = A * B
5:  E = B + 12
6:  F = D / C
```

(a) Code segment



(b) GURRR

Figure 1: Example of GURRR

over-utilize and under-utilize resources, respectively, satisfying the Resource Usage property. Finally, the number of instances of a resource required by a region is stored in the region node. This number can be compared to the number of instances of the resource available for allocation to the region to determine if the region is executable. Due to the hierarchical nature of GURRR, the program is executable if the root region is executable, satisfying the Executability property.

At times during the measurement of resource requirements and use of GURRR by the compiler back-end, it is convenient to consider only subsets of the information provided by GURRR. We identify four combinations of subsets of nodes and edges commonly used. Each combination is a subgraph composed of selected subsets of nodes and edges.

Definition 3 *Given a graph $G = (N, E)$, the subgraph of G* **induced** *by $N' \subset N$* **with respect to** *$\hat{E} \subset E$ is the graph $G' = (N', E')$, where $E' = \{(u, v) \in \hat{E} : u, v \in N'\}$*

1. The **Control Dependence Graph**, CDG, is the graph induced by $\mathcal{I} \cup \mathcal{R}$ with respect to $\mathcal{C}$.
2. The **Data Dependence Graph**, DDG, is the graph induced by $\mathcal{I} \cup \mathcal{R}$ with respect to $\mathcal{D}$.
3. The **Region DAG** for a region R, *Region*$_{\mathtt{R}}$ DAG, is the graph induced by $\{n | n \in \mathcal{I} \cup \mathcal{R} \cup \mathcal{H}$ and $(\mathtt{R}, n) \in \mathcal{C}\}$ with respect to $\mathcal{D} \cup \mathcal{D}_T \cup \mathcal{T}$. This graph provides all of the information needed to allocate all resources in the region.
4. The **Reuse DAG** for a region R and resource R, *Reuse*$_R$ DAG, is the graph induced by $\{n | n \in \mathcal{I} \cup \mathcal{R}$ and $(\mathtt{R}, n) \in \mathcal{C}\}$ with respect to $\mathcal{U}$ and is used to compute a region's requirements, the excessive sets, and resource holes for resource R.

The CDG and DDG are the same as those found the in IPDG. The Region DAG contains all dependence and resource usage information required for performing local unified resource allocation. The CDG and Region DAGs for other regions may be used when performing various types of integrated global resource allocations. The *Reuse* DAG is typically used only by the resource usage computation algorithms. These algorithms measure the resource requirements, compute the excessive sets, and add the resource hole nodes.

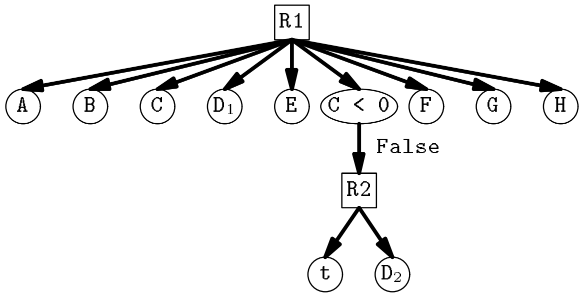
As an example of the various subgraphs, consider the code segment of an `if-then` in Figure 2(a) and assume that the target architecture has a single type of functional unit resource and a single type of register resource. In the subsequent figures edges representing redundant ordering information are removed to aid readability. The control and data dependence subgraphs are shown in Figures 2(b) and 2(c) respectively. The functional unit and register *Reuse* DAGs are shown in Figures 3(a) and 3(b) respectively. The region 2 node, R2, does not occur in the functional unit *Reuse* DAG since its instructions are not executed in parallel with region 1's instructions. The R2 node occurs in the register *Reuse* DAG since the values it computes can be alive simultaneously with some of the values computed in region 1. Since the two values $\mathtt{D}_1$ and $\mathtt{D}_2$ share a register, the R2 node represents the register demand of instruction t. The brlt predicate node does not occur in the register *Reuse* DAG since it does not write to a register. The functional unit *Reuse* DAG for region 1 can be covered by the four allocation chains {C, brlt}, {A, $\mathtt{D}_1$, F, H}, {B, E}, and {G}, indicating a maximum requirement of four functional units to exploit all parallelism in the region. The register *Reuse* DAG can be covered by the six allocation chains {C, F}, {A, $\mathtt{D}_1$, H}, {B}, {E}, {R2}, and {G}, indicating that it is possible for six values to be simultaneously alive.

Figure 3(c) shows the partial schedule for functional units in region 1 imposed by the data and temporal dependencies. Each column represents an allocation chain. There are resource holes before C, after both brlt and E, and before and after G. Since the functional unit is unused for the entire duration of these holes, they are called *free holes*. Instructions $\mathtt{D}_1$, E, and G have slack time in when they can be scheduled. Since the functional units are not needed for the entire time, these nodes exist in *slack holes*. Figure 3(d) shows the region DAG for region 1 with only the functional unit hole nodes. Free and slack hole nodes are marked with FH and SH respectively. A transitive data dependence edge has been added from node C to node F to indicate the transitive dependence caused by nodes t and $\mathtt{D}_2$, which are not in region 1. The largest set of instructions that can be executed in parallel is {C, $\mathtt{D}_1$, E, G}, which would be excessive if the target architecture provided fewer than four functional units.

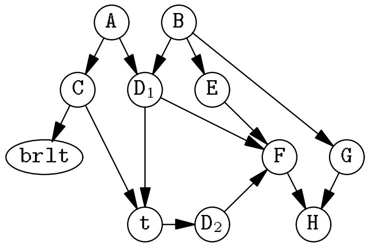
The partial schedule for registers is shown in Figure 3(e). The allocation chain containing R2 does not have any instructions from region 1 and consists of two free holes sepa-

```
 1:  load A
 2:  load B
 3:  C = A * 4
 4:  D1 = A * B
 5:  E = B + 3
 6:  brlt C, 9
 7:  t = D1 + C
 8:  D2 = t * 5
 9:  F = D / E
10:  G = B + 10
11:  H = F + G
```

(a) Code segment



(b) Control Dependence Subgraph



(c) Data Dependence Subgraph

Figure 2: Sample Code and GURRR Dependence Subgraphs

rated by the node R2. Figure 3(f) shows the region 1 DAG without the functional unit holes.

3 Construction of GURRR

The construction of GURRR begins with an IPDG and is performed in a hierarchical manner on the DAG of region nodes resulting from the forward control dependencies. The regions are visited one at a time in a bottom up order and the local components are constructed. A summary of the resource requirements of subregions is used during the construction in the parent region. The resulting global resource requirements are contained in the root region.

Special processing occurs when there are mutually exclusive subregions, such as the `then` and `else` subregions of an `if` statement. In this case, the region containing the `if` statement is only concerned with the maximum requirements of the set of mutually exclusive subregions. The subregions nodes are marked as mutually exclusive and the construction takes the maximum of the requirements for each resource.

The steps in the construction of GURRR for each region are performed in the following order.

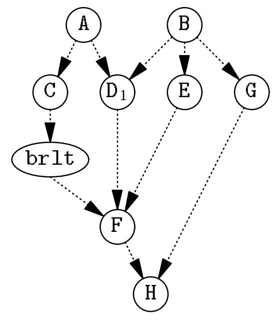
Add transitive data dependence edges: Transitive data dependence edges are added between all instruction and region nodes. The computation of the transitive data dependence edges can be done in graph linear time. In the worst case $O(N^2)$ edges are added. These edges are required for the proper computation of the *Reuse* DAGs.

Build *Reuse* DAGs: The $Reuse_R$ DAG is the instantiation of the relation $CanReuse_R$ for resource R. The $CanReuse_R$ relation identifies the nodes in a region that can safely temporally share an instance of resource R. For nodes A, B, and C in the Data Dependence subgraph, the ordered node pair $(A, B) \in CanReuse_R$ if and only if there is a node C that ends A's use of an instance of R and $C \in Ancestors(B) \cup \{B\}$. Thus $(A, B) \in CanReuse_R$ if and only if B can safely reuse A's instance of R under any schedule allowed by the data dependencies in the data dependence DAG. The $Reuse_R$ DAG is constructed by adding an edge from node A to node B for each $(A, B) \in CanReuse_R$, when both A and B use resource R.

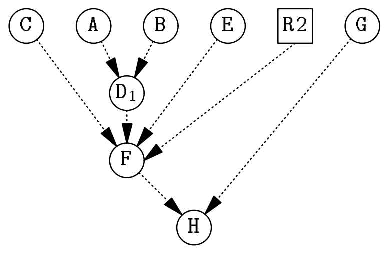
The sets of nodes whose resource can be reused by node n are computed in a forward topological traversal of the DAG using the equation

$$CanReuse_R[n] = avail[n] \bigcup_{P \in predecessors(n)} CanReuse_R[P].$$

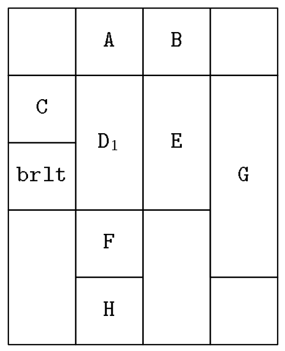
$Avail[n]$ is at most all of n's immediate predecessors whose instances of R can be safely reused by n. The computation of $avail[n]$ is dependent on how each resource is used. There are two classifications of resources based on the duration of a use of the resource. A resource is a *spanning* resource if its use begins during the execution of one instruction, the *defining instruction*, and ends during the execution of a later instruction, the *killing instruction*. A resource is a *non-spanning* resource if its use always begins and ends during the execution of a single instruction. Registers are spanning



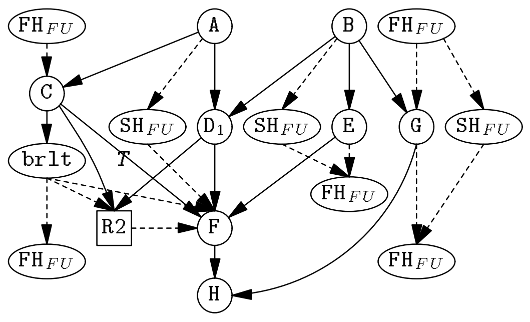
(a) Region 1 Functional Unit Reuse DAG



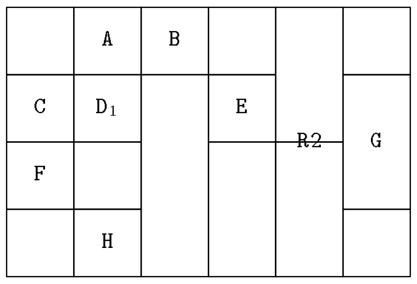
(b) Region 1 Register Reuse DAG



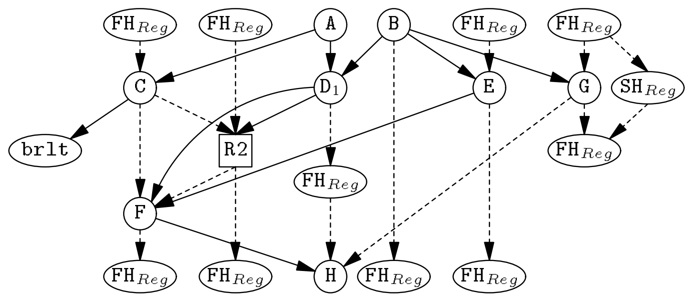
(c) Region 1 FU Schedule



(d) Region 1 DAG with FU Holes



(e) Region 1 Register Schedule



(f) Region 1 DAG with Register Holes

Figure 3: GURRR Resource Usage Information

resources, while functional units are non-spanning resources. For non-spanning resources, $avail[n]$ is the set of n's closest ancestors that use R. Computing $avail[n]$ for spanning resources requires a special component analysis. The identification and analysis of most components can be performed in graph linear time. However, for a few components the analysis is NP-Complete [BGS93]. We formulate the component analysis as a minimal set covering problem and use a graph linear greedy heuristic that has a ratio bound of $ln|X|+1$, where X is the number of nodes in the component [CLR90][1]. The computation of $avail[n]$ and $CanReuse$ are graph linear and the resulting $Reuse$ DAGs contain $O(N^2)$ reuse edges.

[1]In our limited experimentation the components requiring the heuristic had six or fewer nodes. The heuristic always found a precise answer.

Find allocation chains: Allocation chains are chains on the partial order represented by the $CanReuse_R$ relation. The capability of measuring the resource requirements is based on a result by Dilworth, which states that the maximum number of independent elements in a partial order is equal to the number of chains in a minimum decomposition [Dil50]. Thus, the maximum resource requirements of $Reuse$ DAG can be computed by finding the minimum number of allocation chains that cover the $Reuse$ DAG.

Ford and Fulkerson have shown that computing a minimum chain decomposition can be performed using bipartite matching [FF65], in $O(\sqrt{N}E)$ time [HK73]. In practice the matching is performed on the $Reuse_R$ DAG. The matching edges are labeled as such with one in-coming and one outgoing matching edge per instance of R used by the node.

The number of allocation chains for each resource is

type	size	EAT	LAT
Free	$\texttt{LST}_{n_2} - \texttt{EFT}_{n_1}$	$\texttt{EFT}_{n_1}$	$\texttt{LST}_{n_2}$
Slack	*slack*	$\texttt{EST}_{sn_1}$	$\texttt{LFT}_{sn_l}$

Table 1: Computation of hole properties

recorded in the parent region's node. Once the requirements have been measured, the allocation chains are used to compute excessive sets and resource holes.

Find excessive sets: Excessive sets are identified by finding sets of independent instructions whose cardinality is greater than the number of instances of the resource available. Let $|R|$ be the number of instances of resource R available. An excessive set is *grown* using a working list. Each node in the working list is added to excessive set if it is independent of at least $|R|$ nodes on different allocation chains for R. All unexamined nodes that are independent of the excessive node are then added to the working list. The initial node of the work list is located by scanning all nodes until one that meets the excessive test is found. The lists of excessive sets are stored in the region node for use by the allocation phase. This growing process is graph linear in time.

Find resource holes and add hole nodes: Resource holes are found by scanning individual allocation chains and identifying locations where one of two cases exist.

1. A *Free hole* occurs when there are consecutive nodes on the chain where there is a positive amount of time between the Latest Finish Time (`LFT`) of the first instruction and the Earliest Start Time (`EST`) of the second. In this case, the resource instance is completely unused for a period of time between the two instructions.

2. A *Slack hole* occurs when a set of consecutive instructions on an allocation chain is not on a critical path through the region, i.e., the instructions have slack time for scheduling. The slack hole contains these instructions.

Free hole nodes are added between the consecutive nodes surrounding the hole. Slack hole nodes are added between the predecessor of the first instruction or region node in the hole and the successor of the last node in the hole.

Several properties must be computed for each hole found. These properties indicate how the hole can be used to hold instructions. The *size* of the hole indicates how many cycles the resource is unused. The *availability* indicates when the resource is unused, and is represented by the bounds Earliest Available Time (`EAT`) and Latest Available Time (`LAT`).

The computation of both the size and the availability of a hole depends on the type of hole and is summarized in Table 1. Nodes n_1 and n_2 surround a free hole, and nodes sn_1 and sn_l are the first and last nodes in a slack hole. `LST` and `EFT` are the latest start time and earliest finish time of a node, respectively. *Slack* is the slack time of each node in the hole. The hole nodes are annotated with these characteristics. The computation of the `LST` and `EFT` for the instruction and region nodes is graph linear. The location of the holes requires $O(N)$ time, and the worst case number of holes found is $2N$, where N is the number of instruction and region nodes in the region.

4 Applications of GURRR

In this section we describe the use of GURRR for integrating several back-end compiler phases. We first present a technique for integrating global register allocation and instruction scheduling within regions. Next, we examine the integration of global code motions and register allocation. Finally, we integrate parallelizing transformations with resource allocation.

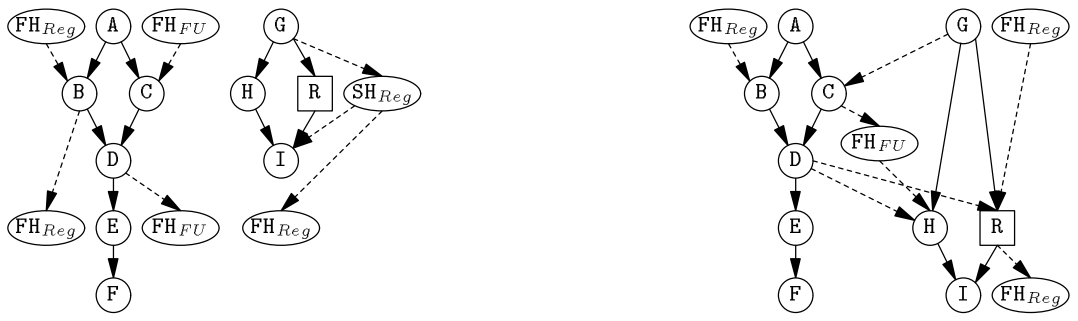
4.1 Global Register Allocation and Instruction Scheduling

Hierarchical register allocation on the Control Flow Graph has been suggested by Callahan and Kennedy [CK91], and on the PDG by Norris and Pollock [NP94]. GURRR supports full integration of hierarchical register allocation and instruction scheduling. The allocations are performed on a region by region basis during a bottom-up traversal of the forward control dependencies.

Because allocation of resources is only difficult when the requirements exceed the availability, our allocation scheme uses a *Measure and Reduce* paradigm. In this paradigm the register and functional unit requirements are measured and the excessive requirements are removed by transformations that introduce additional sequentialization. The measurement process computes the resource usage information found in GURRR. The reduction process selects sets of nodes from an excessive set and attempts to find holes for *all* resources that it excessively demands. By finding holes for all resources needed, unified allocation is achieved. In the case of register holes, additional instructions may need to be inserted with the selected node to perform spilling. There are several cases when different sets of spill instructions need to be inserted, depending on whether the spanning hole is a Free or Slack hole and where the uses of the value(s) computed by the selected nodes occur. Details are given elsewhere [BGS94].

The goal during the selection of holes is to minimize increasing the execution time of the region. Thus, the holes selected should meet two requirements. First, the available time of the holes should overlap with each other, and with the execution range of the node selected from the excessive set. Second, the size of the holes should be large enough for all instructions selected, including any spill code needed. No increase in the critical path length will result if and only if these conditions are met. When the conditions cannot be met *Wedged Insertion* is performed, Wedged Insertion increases the critical path length by stretching the region, and in the process, making holes large enough for the instructions to be inserted. The holes may either be existing holes that were too small or new holes. The location for inserting the wedge is selected to minimize the increase the increase in the critical path length. The selection of the location considers the type of existing and new holes, which affect how much spill code must be inserted to use them.

As an example, consider the region DAG in Figure 4(a). Assume that the region node `R` requires two registers and that the architecture provides two functional units and four registers. Node `G` excessively uses a functional unit. It is forced to use the functional unit hole above `C` by adding a temporal dependence edge from `G` to `C`. Node `H` excessively uses both a functional unit and a register. The register hole following `B` and the functional unit hole following `D` overlap and can be used by `H`, so temporal dependencies are added to force `H` to follow `D`. Node `I` is also inserted in the functional



(a) Region DAG with excessive requirements

(b) Reduced Region DAG

Figure 4: Example of Global Resource Allocation

unit hole following D. To prevent R from using a register excessively while G's value is still alive, R is given the same temporal dependencies as H. The resulting graph is shown in Figure 4(b).

4.2 Global Code Motion

The goal of performing global code motions is to reduce the execution time of the program by exploiting inter-basic block parallelism. Therefore, instructions should not be moved if there are not sufficient resources available at their destination, since no decrease in execution time would result. The control dependence information provided in GURRR naturally allows inter-block motion for blocks with the same control dependencies. However, code motions between regions with different control dependencies are also possible. Some motions can be performed using code duplication or safe speculative execution without hardware support. Other motions require predicated or speculative hardware support. We assume that the code motion algorithm is provided with the sets of instructions that can be moved according to the semantics and hardware support available.

To give priority for resources to the instructions intrinsic to each region, unified register allocation and instruction rescheduling is performed first. Global code motion then uses the remaining resource holes for instructions that it elects to move. With the exception of code duplication, wedged insertion is not performed during global code motion.

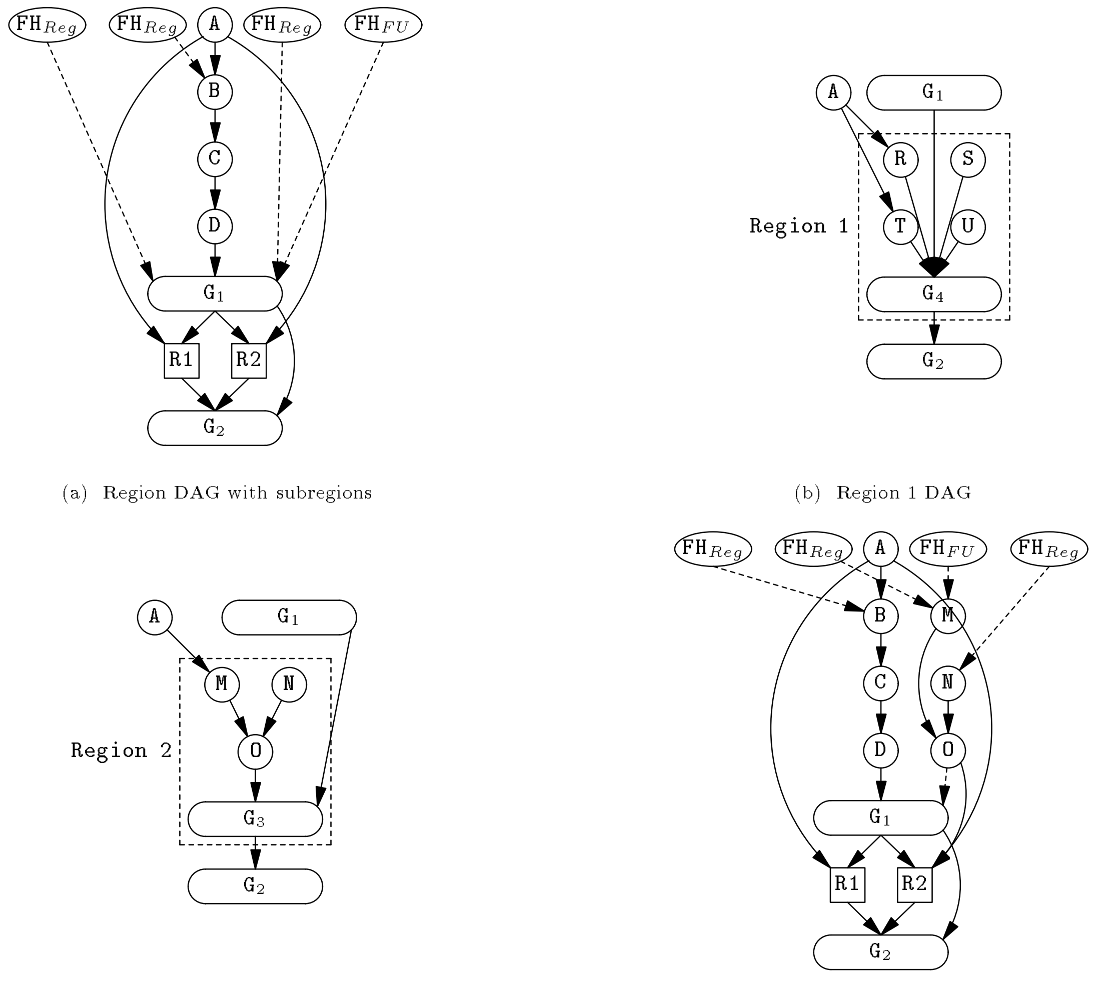
The code motion algorithm must find overlapping resource holes for all resources that each moved instruction uses. There are several possible strategies for performing global code motion: 1) instructions can be moved individually, 2) instructions can be moved in sets that each decrease the critical path of the source region, or 3) a single set of instructions to be moved can be selected by measuring their requirements and estimating their hole usage. It is generally advantageous to move large sets of instructions, so that the amount of spill code generated is minimized. In the first case each instruction is treated as a node selected from an excessive set. Holes are found for all of its required resources. The other cases are handled by placing the set of instructions in the destination region and using the Measure and Reduce paradigm.

The difference between the second and third approaches is in the selection of sets and the method of termination. In the second approach, successive sets are selected and moved until moving a set would cause an increase in the destination region's execution time. In the third approach only one set of instructions is selected and moved, based on resource usage estimates. The total requirements of the selected instructions can be determined simply by counting the number of allocation chains that they are on. The total size of holes required for each subchain is also directly available. The estimates consider how many instructions can be inserted into each hole while preserving dependencies.

As an example, consider the region DAG in Figure 5(a). The G_i nodes represent groups of instruction nodes. Assume that the architecture has two functional units and four registers, and that group G_1 uses all resources. The region DAGs for R1 and R2 are shown in Figures 5(b) and 5(c), respectively. Nodes from the parent region are added to show the inter-region data dependencies. Global code motion can be performed on either the critical sets {R, S} and {T, U} in region 1 or the critical sets {M, N} and {O} in region 2. The critical set {R, S} in region 1 can share one functional unit, but requires two registers. There are insufficient registers to also move the critical set {T, U}. The critical sets in region 2 can share one functional unit and require two registers. The critical sets from region 2 are moved up because they result in a larger decrease in critical path length. The resulting region DAG is shown in Figure 5(d). A temporal dependence is added from O to G_1 to ensure that a register is freed by O in time for G_1 to use it. Even if sufficient registers were available for both of region 1's critical set, it is not advantageous to move it because functional unit availability only allows one instruction from the second critical to be moved, at no additional improvement.

4.3 Transformation Requirements Prediction

The resource requirements of a program can be used to estimate the impact of potential transformations. When several code transformations are available, it is advantageous to select the one that exposes just enough parallelism. If too much parallelism is exposed the compiler must remove the excess during scheduling and register allocation. Heuristics can be developed that given a transformation, use the number of allocation chains covering the portions of the program transformed to estimate what the new resource requirements would be. A second approach is to use incremental updating of GURRR to compute the resource requirements resulting from the transformations.



(a) Region DAG with subregions

(b) Region 1 DAG

(c) Region 2 DAG

(d) Resulting Region DAG

Figure 5: Example of Global Code Motion

As an example, consider the body of a loop shown in Figure 6(a), and assume an architecture with four functional units. The loop requires two functional units. Loop unrolling is a parallelizing transformation that allows overlapping of successive iterations. It is desirable to perform only as much unrolling as will expose parallelism that can be exploited by the target architecture. The unrolling process is repeatly performed until there is no more decrease in the number of cycles required to execute the unrolled iterations. If the current resource requirements were used in a heuristic that considered the data dependencies, i.e., two functional units, two iterations could be overlapped without exceeding the four available functional units. The result would be the first two diamonds in Figure 6(b). The length of the resulting block would be 4 cycles, an improvement over 6 cycles required for iterations that were not overlapped. If another iteration is unrolled and overlapped with the earlier iterations and the resource requirements were updated, three iterations can be overlapped for further improvement. The resulting set of instructions $\texttt{B}_1$, $\texttt{T1}_2$, $\texttt{T2}_2$, and $\texttt{A}_3$ can be used as the kernel of a software pipeline [Lam88, AN88b].

5 Incremental Updating of GURRR

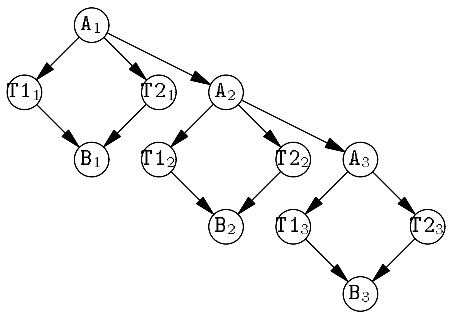
GURRR is able to reflect changes in resource requirements resulting from the transformations applied to the program. The brute force approach is to recompute all information from scratch after each transformation is applied. This can be a costly approach, and it does not provide any support for predicting the impact of a transformation. It would be useful to be able to estimate the impact of a possible transformation on the resource requirements. In this section we sketch techniques for incrementally updating GURRR.

In previous work on specifying transformations a basic set of program edits to describe the transformations has been used [WS91, Dow94]. We define the following set of *Standard Edit Functions* (SEFs), which apply a transformation to the elements of a PDG.

`Add`*element*	Create a new element
`Delete`*element*	Delete an element
`Copy`*element*	Create a new element and copy label information of an existing element

```
DO i = 2 to 10
1:  A[i] = A[i-1] + 10
2:  T1 = A[i] - 1
3:  T2 = A[i] + 1
4:  B[i] = T1 * T2
ENDDO
```

(a) Loop Body



(b) DAG

Figure 6: Loop candidate for unrolling

`Move`*element*	Delete and recreate an element, preserving label information
`Modify`*element*	Change the label information

There are two sets of the above operations, one for nodes and one for edges, giving a total of ten SEFs. Since nodes will never be added without corresponding edges, and the edge SEFs can be viewed as combinations of adding and deleting edges, we consider only the `AddEdge` and `DeleteEdge` SEFs.

The computation of the *CanReuse* relation is graph linear. The updating of $avail[n]$ information is limited to the nodes affected by the `AddEdge` and `DeleteEdge` SEFs. The updated information is then propagated through the region DAG. The *Reuse* DAG is updated by adding and deleting edges corresponding to the nodes inserted and removed in the *CanReuse* relation.

The matching algorithm used to compute allocation chains is incremental by nature; each matching is a partial solution and new matchings are added by finding augmenting paths. Thus, the modified *Reuse* DAG with edges deleted and added can be used as partial solution. The complexity for this solution is $O(\sqrt{m}E)$, where m is the number of chains in the initial partial solution. An alternative approach can find only unit length augmenting paths in graph linear time, possibly introducing some imprecision.

Updating of the excessive sets is performed in two steps. First, the nodes in the existing excessive sets are tested to see if they are still in parallel with an excessive number of other nodes. This step can be limited to the nodes that have had edges added to them. Second, nodes not in the excessive sets are tested to see if they now should be. The initial set of nodes considered in this step can be limited to those that have had edges removed.

Transformations can affect holes by creating new ones, removing existing ones, and by changing their characteristics. All of these changes can be found by examining each node whose `EST` and/or `LFT` has changed. However, the nature of the matching algorithm used to find the allocation chains can cause unchanged holes to migrate between allocation chains. The sequential edges used to place the hole nodes in the region DAGs can be updated to reflect the migrations in linear time in the number of hole nodes.

6 Related Representations

Traditionally, compilers have used the Control Flow Graph (CFG) as the intermediate representation of the program. The CFG is used to collect a variety of information, including dataflow dependencies and live value ranges [ASU86]. Extensions to the CFG such as Traces [Fis81] and Super Blocks [HMC$^+$93] have been developed in an effort to support global code motion. The CFG is also traditionally used to construct register interference graphs.

The Program Dependence Graph (PDG) combines Control and Data dependence information to simplify many transformations [FOW87]. Control dependencies are used to identify *regions* of instructions that execute under the same conditions. Regions support more powerful global code motion techniques than are possible on the CFG [GS90, BR91].

Static Single Assignment (SSA) form uniquely assigns names to each definition of a variable [RWZ88]. The use of unique names simplifies constant propagation and other analyses [AWZ88]. SSA was originally formulated on the CFG but has been incorporated in PDG based representations [BMO90].

The Program Structure Tree (PST) is a hierarchical representation that can be used by divide-and-conquer algorithms to speedup dataflow analysis and computation of SSA. The PST does not directly identify the control dependencies used by region based global code motion algorithms.

A number of representations allowing direct interpretation have been proposed, including the Dependence Flow Graph [PBJ$^+$91, JP93] and Value Dependence Graph [WCES94], but do not contain control dependence information. The Program Dependence Web [BMO90] is an interpretable representation that places a variation of SSA form on the PDG. Our current research has not examined uses requiring an interpretable representation and has concentrated on global code motion algorithms that exploit control dependence information.

None of the representations mentioned satisfactorily meet the unified representation properties. Register demands are traditionally handled separately from functional units. Thus, the representations do not satisfy the Unified Representation property. Since these representations are based on some form of a data dependence graph, which is quite similar to the $Reuse_{FU}$ DAG, they can be considered to satisfy the Measurability property for functional units. However, actual measurements, resource usage levels, and executability are not computed.

Typically, a graph coloring approach is used to address register demands. The Register Interference Graph is constructed to indicate which variables compete for registers.

The PDG can be used to build a register interference graph that more accurately reflects overlapping live ranges in parallel programs [Pin93, NP93, AEBK94].

The interference graph provides a method for measuring register demands. In practice, an inaccurate measure of demands is computed by counting the number of other values that a given value interferes with. A more accurate method could find cliques in the interference graph. Using this measurement, a limited form of register usage levels could be computed, as well as executability. However, since the interference graph is not integrated with functional unit demands, it does not indicate the impact of allocation decisions on the parallelism of a program or the length of the critical path. Without some form of resource usage information the Executability property cannot be satisfied

GURRR uses results from our earlier work. In the Unified ReSource Allocator (URSA) we developed the Measure and Reduce paradigm and the algorithms to measure resource requirements, i.e., allocation chains, and find excessive sets [BGS93]. URSA was designed to operate on large basic blocks, such as those resulting from Trace Scheduling. Simple transformations were used to reduce the excessive resource requirements by introducing temporal dependencies to sequentialize the excessive demands. Resource Spackling introduced the notion of resource holes and used the resource usage information to develop reduction transformations that moved excessive resource demands to resource holes [BGS94]. These reduction transformations are used for local scheduling and global code motion. Resource Spackling was designed to be a framework that could combine the resource usage information with several global code motion mechanisms and their corresponding representations. The representations and mechanisms include the PDG using Region Scheduling [GS90] and the Control Flow Graph using either Trace Scheduling [Fis81] or Percolation Scheduling [AN88a]. The resource usage information computed did not include hierarchical resource requirements.

7 Summary

We have presented the Global Unified Resource Requirements Representation (GURRR) for use in integrating phases of a compiler for instruction level parallelism. GURRR is a collection of resource usage information superimposed on a instruction level PDG (IPDG). The additional information consists of new types of nodes, edges, and annotations. Two new subgraphs are introduced to view useful combinations of information. The resource *Reuse* DAGs indicate which instructions can temporally share instances of a resource, and are used to compute each regions' requirements for each resource. These resource requirements measurements are used to identify locations where resources are over- and under-utilized, called Excessive Sets and Resource Holes, respectively. Node annotations include indication of any excessive sets that a node is a member of, descriptions of holes that the node identifies, and holes that the node may use. Region nodes also contain summaries of the resource requirements, excessive sets, and resource holes for the nodes in its region. Region DAGs provide all information needed to perform unified resource allocation in a region.

We outline the computation of all information in GURRR and demonstrate its usefulness in three areas of integration. Register allocation and instruction scheduling are fully integrated in a hierarchical allocator. The integration is achieved by simultaneously allocating registers and functional units to each instruction. Allocation decisions for registers and functional units are based on their impact on the resulting length of the critical path. This technique is then used in a global code motion algorithm to move instructions only if all resources they require are available, ensuring that the execution time is reduced. We discuss predicting the impact of transformations on a program's resource requirements. This capability allows the compiler to select which transformations to apply so that it does not have to remove excess requirements that some transformations may introduce. Finally, we show how GURRR can be incrementally updated to reflect changes in resource usage due to allocation decisions or program transformations.

We have implemented GURRR in the University of Pittsburgh's experimental compiler tool, `pdgcc`. `Pdgcc` is a C compiler front-end which performs dependence analysis and generates intermediate code in the form of PDGs. We are currently implementing the applications in section 4.

References

[AEBK94] Wolfgang Ambrosch, M. Anton Ertl, Felix Beer, and Andreas Krall. Dependence-conscious global register allocation. In Jürg Gutknecht, editor, *Programming Languages and Systems Architecture*, pages 125–136, Zürich, April 1994. Springer LNCS 782.

[AN88a] Alexander Aiken and Alexandru Nicolau. A development environment for horizontal microcode. *IEEE Trans. on Software Engineering*, 14(5):584–594, 1988.

[AN88b] Alexander Aiken and Alexandru Nicolau. Optimal loop parallelization. In *Proc. of Sigplan '88 Conf. on Programming Language Design and Implementation , ACM Sigplan Notices*, volume 23, pages 308–317, 1988.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison Wesley, Reading, Massachusetts, 1986.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. In *Conf. Rec. 15th ACM Symp. on Prin. of Programming Languages*, pages 1–11, 1988.

[BGS93] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. URSA: A Unified ReSource Allocator for registers and functional units in VLIW architectures. In *Proc. of IFIP WG 10.3 Working Conference on Architectures and Compliation Techniques for Fine and Medium Grain Parallelism*, pages 243–254, 1993. (also available as University of Pittsburgh Computer Science Department Technical Report 92-21).

[BGS94] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Resource Spackling: A framework for integrating register allocation in local and global schedulers. In *Proc. of IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pages 135–146, 1994. (also available as University of Pittsburgh Computer Science Department Technical Report 94-09).

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proc. of Sigplan '90 Conf. on Programming Language Design and Implementation*, pages 257–271, 1990.

[BR91] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proc. of Sigplan '91 Conf. on Programming Language Design and Implementation*, pages 241–255, 1991.

[CK91] David Callahan and Brain Koblenz. Register allocation via hierachical graph coloring. In *Proc. of Sigplan '91 Conf. on Programming Language Design and Implementation*, pages 192–203, 1991.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Mass., 1990.

[Dil50] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annuals of Mathematics*, 51:161–166, 1950.

[Dow94] Chyi-Ren Dow. Pivot: A program parallelization and visualization environment. Technical Report Technical Report 94-22, Ph.D. Dissertation, University of Pittsburgh, Computer Science Department, 1994.

[FF65] L. R. Ford and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, N.J., 1965.

[Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Computers*, C-30(7):478–490, 1981.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. and Systems*, 9(3):319–349, 1987.

[GH88] James R. Goodman and Wie-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *Proc. of ACM Supercomputing Conf.*, pages 442–452, 1988.

[GS90] Rajiv Gupta and Mary Lou Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Trans. on Software Engineering*, 16(4):421–431, 1990.

[HK73] John E. Hopcroft and Richard M. Karp. An $N^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM JOURNAL of Computing*, 2(4), 1973.

[HMC+93] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The Superblock: An effective technique for VLIW and superscalar compliation. In *The Journal of Supercomputing*, volume A, pages 229–248, 1993.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proc. of Sigplan '93 Conf. on Programming Language Design and Implementation*, pages 78–89, 1993.

[Lam88] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of Sigplan '88 Conf. on Programming Language Design and Implementation , ACM Sigplan Notices*, volume 23, pages 318–328, 1988.

[NG93] Qi Ning and Guang R. Gao. A novel framework of register allocation for software pipelining. In *Conf. Rec. 20th ACM Symp. on Prin. of Programming Languages*, pages 29–42, 1993.

[NP93] Cindy Norris and Lori Pollock. A scheduler-sensitive global register allocator. In *Proc. of Supercomputing '93*, pages 804–813, 1993.

[NP94] Cindy Norris and Lori L. Pollock. Register allocation over the program dependence graph. In *Proc. of Sigplan '94 Conf. on Programming Language Design and Implementation*, pages 266–277, 1994.

[PBJ+91] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *Conf. Rec. 18th ACM Symp. on Prin. of Programming Languages*, pages 67–78, 1991.

[Pin93] Shlomit S. Pinter. Register allocation with instruction scheduling: A new approach. In *Proc. of Sigplan '93 Conf. on Programming Language Design and Implementation*, pages 248–257, 1993.

[RLTS92] B. R. Rau, M. Lee, P.P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proc. of Sigplan '92 Conf. on Programming Language Design and Implementation*, pages 283–299, 1992.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conf. Rec. 15th ACM Symp. on Prin. of Programming Languages*, pages 12–27, 1988.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Conf. Rec. 21st ACM Symp. on Prin. of Programming Languages*, pages 297–310, 1994.

[WS91] Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. In *Proc. of Sigplan '91 Conf. on Programming Language Design and Implementation*, pages 120–129, 1991.