Integrated Instruction Scheduling and Register Allocation Techniques*

David A. Berson[1] Rajiv Gupta[2] Mary Lou Soffa[2]

[1] Intel Corporation, Microcomputer Research Lab, 2200 Mission College Blvd., Santa Clara, CA 95052

[2] Dept. of Computer Science, Univ. of Pittsburgh, Pittsburgh, PA 15260, USA

Abstract. An algorithm for integrating instruction scheduling and register allocation must support mechanisms for **detecting excessive** register and functional unit demands and **applying reductions** for lessening these demands. The excessive demands for functional units can be detected by identifying the instructions that can execute in parallel, and can be reduced by scheduling some of these instructions sequentially. The excessive demands for registers can be detected on-the-fly while scheduling by maintaining register pressure values or may be detected prior to scheduling using an appropriate representation such as parallel interference graphs or register reuse dags. Reductions in excessive register demands can be achieved by live range spilling or live range splitting. However, existing integrated algorithms that are based upon mechanisms other than register reuse dags do not employ live range splitting. In this paper, we demonstrate that for integrated algorithms, register reuse dags are more effective than either on-the-fly computation of register pressure or interference graphs and that live range splitting is more effective than live range spilling. Moreover the choice of mechanisms greatly impacts on the performance of an integrated algorithm.

1 Introduction

The interaction of instruction scheduling and register allocation is an important issue for VLIW and superscalar architectures that exploit significant degrees of instruction level parallelism (ILP). Register allocation and instruction scheduling have somewhat conflicting goals. In order to keep the functional units busy, an instruction scheduler exploits ILP and thus requires that a large number of operand values be available in registers. On the other hand, a register allocator attempts to keep the register pressure low by maintaining fewer values in registers so as to minimize the need for generating spill code.

If register allocation is performed first, it limits the amount of ILP available by introducing additional dependences between the instructions based on the temporal sharing of registers. If instruction scheduling is performed first, it can create a schedule demanding more registers than are available, causing more

* Supported in part by NSF Grants CCR-9402226 and CCR-9808590 to the Univ. of Pittsburgh.

work for the register allocator. In addition, the spill code subsequently generated must be placed in the schedule by a post-pass cleanup scheduler, degrading the performance of the schedule. Thus, an effective solution should integrate register allocation and instruction scheduling.

An integrated approach must provide mechanisms for *detecting* excess demands for both functional unit and register resources and for *reducing* the resource demands to the level supported by the architecture. Excessive demands for functional units due to a high degree of ILP are reduced by the instruction scheduler by scheduling the execution of independent, and thus potentially parallel, instructions sequentially. Excessive demands for registers cannot always be reduced through sequentialization alone and may further require the saving of register values in memory through live range spilling or live range splitting.

In this paper we demonstrate that the performance of an integrated algorithm is greatly impacted by the mechanism it uses to determine excessive register demands and the manner in which it reduces the register demands. Excessive register demands can be determined by maintaining register pressure during scheduling, constructing a parallel interference graph, or by constructing register reuse dags. Reduction can be achieved through live range spilling or live range splitting. Our results show that excessive register demands can be best determined using register reuse dags and reduction is best achieved through live range splitting. However, none of the existing integrated algorithms are based upon these mechanisms. We implemented newly developed integrated algorithms as well as existing algorithms to obtain the above results as follows.

- The *on-the-fly* approach (IPS) developed by Goodman and Hsu [11] performs local register allocation within extended basic blocks during instruction scheduling. It tracks *register pressure* to detect excessive register demands and uses *live range spilling* to reduce register pressure. We extended this technique to incorporate *live range splitting* (ILS). Based upon the performances of the original and extended versions of the algorithm we conclude that live range splitting is far superior to live range spilling when developing an integrated resource allocator.
- The *parallel interference graph* approach developed by Norris and Pollock [14] uses an extended interference graph to detect excessive register demands and guide schedule sensitive register allocation (PIR). The reduction in register demands is achieved through *live range spilling*. We modified this technique to incorporate the use of the register reuse dag in place of the interference graphs for detecting excessive register demands (RRD). Variations of priority functions for selecting candidate live ranges for spilling are also considered. By comparing the performances of the above algorithms we conclude that register reuse dags are superior to interference graphs.
- The *unified resource allocation* (URSA) approach developed by us is based upon the *measure-and-reduce* paradigm for both registers and functional units [4]. Using the *reuse dags*, this approach identifies *excessive sets* that represent groups of instructions whose parallel scheduling requires more resources than are available [1]. The excessive sets are then used to drive re-

ductions of the excessive demands for resources. *Live range splitting* is used to reduce register demands. This algorithm performs better than the algorithms based upon the on-the-fly approach and interference graphs and also has the lowest compilation times.

The table given below summarizes the integrated algorithms implemented in this work.

Register Pressure Computation	Live Range Spilling	Live Range Splitting
On-the-fly	IPS	ILS
Parallel Interference Graph	PIR	-
Register Reuse DAG	RRD	URSA

The significance of integration is greatly increased in programs where the register pressure is high. Thus when compiling programs for VLIW architectures, or other types of multiple issue architectures, the need for integration is the greatest. In comparing the above algorithms, we experimentally evaluated the integrated algorithms using a 6 issue architecture. Previous studies have been limited to single issue pipelined machines and therefore do not reveal the true significance of integration. In our algorithms, both instruction scheduling and register allocation are performed hierarchically over the program dependence graph (PDG) [10]; that is, each algorithm traverses the control dependence graph in a bottom-up fashion, performing integrated instruction scheduling and register allocation in each region and then using the results at the next higher control dependence level.

In section 2 we provide an overview of important issues that an integrated algorithm for instruction scheduling and register allocation must address. In section 3 we describe algorithms that perform on-the-fly register allocation and study the effect of live range spilling and splitting on performance. In section 4 we evaluate an algorithm based upon a parallel interference graph approach and compare it with one that uses register reuse dags. In section 5 we describe algorithms based upon the unified resource allocation approach which employs both register reuse dags and live range splitting. We conclude by summarizing the main results of this work in section 6.

2 Issues in Integrating Register Allocation with Instruction Scheduling

Each of the integrated instruction scheduling and register allocation algorithms must support mechanisms for detecting excess requirements for functional units and registers as well as techniques for reducing these requirements to the levels supported by the architecture. In addition, the order in which reductions for functional units versus registers are applied may differ from one technique to another. We first discuss a variety of detection and reduction methods that have been proposed and then we briefly describe the specific choices made by algorithms implemented in this study.

Excessive requirements for *functional units* arise when the degree of parallelism identified in a program segment is found to be greater than the number of functional units available. The excess parallelism may be identified either *on-the-fly* while the schedule is being generated or *precomputed* prior to the start of instruction scheduling. An example of the former approach is a list scheduler which can, at any point in time during scheduling, identify excess parallelism by simply examining the ready list for the number of operations that are ready for scheduling. An example of the latter approach is one in which an acyclic data dependence graph is constructed for a code segment prior to scheduling and examined to identify the maximum degree of parallelism.

Reductions of functional unit resources are performed by sequentially scheduling some of the operations that are able to execute in parallel. Reductions can be performed either on-the-fly or prior to scheduling. A priority based list scheduler faced with excess parallelism may first on-the-fly choose the nodes with higher priority for scheduling while delaying the scheduling of other ready nodes. Reductions can also be performed prior to scheduling by introducing sequentialization edges in an acyclic data dependence graph to reduce the maximum degree of parallelism in the graph.

Excessive requirements for *registers* arise when the number of values that are live exceed the number of registers available. Similar to functional units, the excess register requirements for registers can be detected *on-the-fly* during scheduling or *precomputed* prior to scheduling. A list scheduler will identify excess register requirements when it tries to schedule an instruction and finds that no register is free to hold the result of the instruction. Excess register requirements can be precomputed using two different methods. The first method, used by register allocators based on graph coloring, identifies excessive register demands by finding uncolorable components in the *parallel interference graph*. Another method constructs a directed acyclic graph, called the *register reuse dag*, in which an edge is drawn from one instruction to another if the latter is able to reuse the register freed by the former. By finding the maximum number of independent instructions in the register reuse dag, the excessive register demands are identified. A set of instructions identified to require more registers than are available is said to form an *excessive set*.

There are a number of register reduction techniques available. The first is *sequentialization* which orders instructions that can be executed in parallel so that the instructions can use the same register. This reduction technique is not always applicable due to other dependences in the program. The second reduction technique is *live range spilling*, where a store instruction is inserted immediately after the definition of the value. A load instruction is then inserted before every use of the value. This approach results in a large number of loads from memory; however, it can always be performed and removes many interferences. The third reduction technique is *live range splitting*, which tries to reduce the number of load instructions by having several close uses of the value share a single load. In determining the uses that should share a register, the register allocator must ensure that the register pressure does not exceed the limit imposed by the architecture. In the case where the instructions have not been scheduled yet, it is difficult for the register allocator to know how "close" several uses are, or if by sharing a load, they will result in competing with other live ranges. To our knowledge, none of the previously developed integrated techniques use live range splitting. Finally by combining the introduction of sequential dependences with live range splitting, a special form of live range splitting can be performed in cases where neither sequentialization nor live range splitting alone would be feasible or result in a reduction of register pressure.

In developing an algorithm that integrates instruction scheduling and register allocation, the selected register allocation and scheduling techniques to detect and reduce the requirements must cooperate in some way. No integration means that the heuristics for register allocation and scheduling are performed independently of one another in separate phases. One approach to integration is to allocate register allocation and functional units simultaneously in one phase, resulting in a fully integrated system. Another approach is to allocate the resources separately, but use information about the allocation of the other resource. There are various strategies that can be used to order the allocation phases. One strategy is to allocate all of resources of one type in one phase and then allocate the other resource in a subsequent phase, passing information from one phase to the other. Another ordering would be to interleave the allocation heuristics. Thus, some resources of one type are allocated and then some of the other type are allocated. The important component of either approach is the information about one resource that is used during the allocation of the other resource.

3 Live Range Spilling vs Live Range Splitting

Typically a list scheduler uses the *heights* of instructions in the dependence DAG to prioritize the scheduling. In this technique, to reduce excessive register demands, *register pressure* is continuously tracked during instruction scheduling and used in conjunction with instruction heights to guide scheduling. Thus, the excessive demands for both resources are reduced by the scheduler in the process of selecting instructions for scheduling. If register pressure exceeds the maximum number of registers available, register spilling is required.

The two algorithms based upon this approach that were implemented differ in their treatment of excessive register requirements. The IPS algorithm proposed by Goodman and Hsu [11] addresses the excessive requirements for registers through *live range spilling* which is carried out during a separate pass following the scheduling prepass using extended basic blocks. The ILS algorithm developed by us extends IPS by performing register allocation hierarchically on a PDG and eliminating a need for separate spilling pass by performing *live range splitting* during instruction scheduling. Next we describe the two algorithms in greater detail.

In IPS the list scheduler alternates between two states. In the first state, register pressure is low and the scheduler selects instructions to exploit ILP based upon their heights. When the register pressure crosses a threshold, the scheduler switches to a second state that gives preference to instructions that reduce register pressure, possibly sacrificing opportunities to exploit ILP in the process. Additionally, no spilling of values is performed. When register pressure falls back below the threshold, the first state is reentered. In this manner scheduling and allocation are integrated. IPS attempts to sequence live ranges to reduce live range interferences. This reduction is accomplished by giving preference to scheduling instructions that kill live ranges prior to ones that only start new ones when register pressure reaches a specified threshold. If the scheduler is unable to select instructions in a manner that keeps the register pressure below the maximum allowed by the architecture, then live range spilling is unavoidable. A postpass register allocation via coloring is performed to handle any register allocation problems that the scheduler is unable to address. This register allocator uses a traditional priority based coloring approach to select candidate live ranges for spilling [8].

The ILS algorithm eliminates the need for the spilling postpass by using *live range splitting* during instruction scheduling to ensure that the register pressure never exceeds the maximum allowable value. This approach requires that ILS be applied hierarchically in a bottom-up fashion so that the live ranges that extend across child regions are also considered in computing the register pressure. ILS maintains a list of all values that are alive in the cycle currently being scheduled. When ILS detects that register pressure is high, and there are no ready instructions that reduce the number of currently active live ranges, it selects a live range for splitting. ILS injects a store instruction into the ready list and a load instruction dependent on the store into the not-ready list. ILS then moves the dependencies of all unscheduled uses of the value from the original definition to the injected load. In this manner ILS essentially performs live range splitting. The priority function used for selecting a live range for splitting gives preference to the live range whose earliest subsequent use is farthest from the current instruction cycle being scheduled. To avoid useless loads of values the priority functions are designed to not schedule injected load instructions unless it can be guaranteed that at least one of the dependent uses can also be scheduled. The incorporation of live range splitting into ILS creates a powerful and complete single pass allocation algorithm for both registers and functional units.

Next we present results of experiments that compare the performances of ILS and IPS algorithms. Performances of all algorithms are presented in terms of the speedups they achieve for a 6 issue machine in comparison to a base algorithm for a single issue machine. The base algorithm was chosen to be the interference graph based algorithm since it performed the worst of all the algorithms. Since the objective of the experiments is to see how well various algorithms perform under high register pressure, the algorithms were executed for a machine with varying number of registers. In computing the speedups achieved by any algorithm over the base algorithm, the same number of registers were provided to both algorithms. By doing so the results that are obtained demonstrate the impact of integration capability of an algorithm on the effectiveness with which parallelism is exploited.

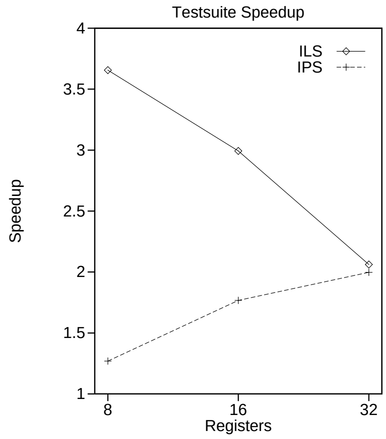


Fig. 1. Comparison of IPS and ILS for a 6 issue architecture.

The performances of IPS and ILS are shown in Figure 1. As we can see, our ILS algorithm performs much better than the IPS approach. After analyzing the code generated by the two algorithms we observed that this difference in performance was attributable to significantly greater amounts of spill code introduced by IPS. Thus, we conclude that an algorithm such as ILS that incorporates live range splitting performs better than an algorithm such as IPS that is based only on spilling. This result is not entirely unexpected as spilling can be viewed as a special case of splitting. The results also show that the difference in the performance of ILS and IPS decreases as greater numbers of registers are made available. This trend indicates that the effectiveness of integration strategy has a greater impact on performance for higher register pressures.

4 Parallel Interference Graphs vs Register Reuse Dags

A more sophisticated approach for global register allocation is based upon the coloring of interference graphs [9, 8]. This approach was extended to make the process of register allocation schedule sensitive through the construction of a *parallel interference graph* [14, 15]. The algorithm (PIR) we implemented is based on a parallel interference graph proposed by Norris and Pollock [14] and uses a Chaitin [8] style register allocator which relies upon live range spilling. The parallel interference graph represents all interferences that can occur in legal schedules.

The interference graph is constructed by adding interference edges between nodes representing live ranges that may overlap in the current region or in the child regions of the current region. The interference graph is simplified by removing all nodes that are incident on fewer edges than the number of registers available. The remaining nodes are then processed to reduce the register requirements. Reductions are achieved using both *sequentialization* and *spilling*. While *live range splitting* has been incorporated into traditional coloring based register allocators [9], it has not been incorporated in schedule sensitive allocators based upon parallel interference graphs due to the lack of a complete ordering of the instructions. Without a complete ordering, it cannot be guaranteed that a particular splitting of a live range will reduce the number of interferences. Therefore the *splitting* reduction is not used by existing algorithms.

The order in which nodes are considered for reduction is based upon cost functions that prioritize the nodes. After applying a reduction, the interference graph is recomputed and the process is repeated until no more reductions are required. At this point, all nodes can be successfully colored; that is, register allocation is now complete. In the process of coloring, the instructions are partially scheduled through the application of sequentialization reductions. A postpass list scheduler is run as a last step to produce the final code schedule. In our implementation of PIR, the ILS scheduler (with register allocation turned off) was used for this purpose.

The cost functions that prioritize the nodes compute the cost for both spilling the value and for sequentializing the live range after all uses of another live range. The costs are computed in terms of the effect of the transformations on the critical path length. The minimum of these two costs is used as the priority and the corresponding reduction method is recorded in case the node is selected for reduction.

Sequentialization of a live value defined by D_1 is performed by finding a second value D_2 which interferes with D_1 and then introducing temporal dependences from all uses of D_2 to definition D_1. The cost of sequentialization reduction is computed using the following formula:

$$Cost_{seq} = \frac{max_{u \in Uses(D_2)}(u.EST + D_1.LST) \ominus cpl}{NumInterferences}$$

where $u.EST$ is the earliest start time of the use instruction u, $D_1.LST$ is the latest start time of definition D_1, cpl is the critical path length of the region containing D_1, $Uses(D_2)$ is the set of uses of D_2, and the symbol $\ominus$ represents *floored subtraction* function ($a \ominus b = if\ a > b\ then\ a - b\ else\ 0$).

Two different functions were used to compute the cost of spilling, measured by the effect on the critical path lengths. The first cost function considers the increase in critical path length for a given region as the number of loads and stores that are required in the region. The total increase in critical path length was computed by summing together the product of the increase in length and the execution count of all relevant regions. The second priority function considers the slack time in scheduling spill code in computing the increase in critical path length. The slack time of an instruction is the difference between the earliest time and the latest time at which the instruction can be scheduled. The motivation behind this cost function is that high slack times reduce the likelihood of an increase in critical path length. The formulas for computing the spill costs based upon the above approaches are given below:

$$Cost_{spill} = \frac{StoreCost \times def.ExecCnt + \sum_{u \in uses} LoadCost \times u.ExecCnt}{NumInterferences}$$

$$SCost_{spill} = \frac{(StoreCost \ominus def.Slack) \times def.ExecCnt + \sum_{u \in uses} LoadCost \times (u.ExecCnt \ominus u.Slack)}{NumInterferences}$$

where $StoreCost/LoadCost$ is the cost in cycles to execute a store/load instruction, $i.ExecCnt$ is the execution count for the region containing instruction i, $i.Slack$ is the slack time of instruction i, $NumInterferences$ is the number of other live ranges with which the spilled value interferes, and the symbol $\ominus$ represents the *floored subtraction* function.

We evaluated the register coloring approach using both of the above cost functions. The algorithm PIR uses the first spill cost function and the algorithm $SPIR$ uses the second spill cost function that incorporates slack times. The results of these evaluations are shown in Figure 2.

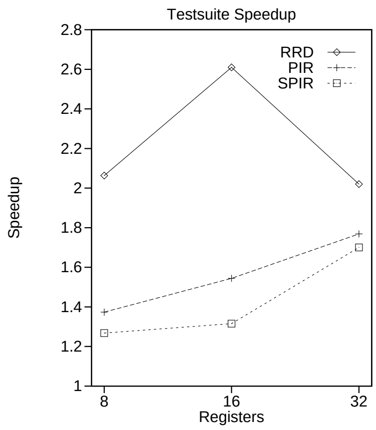


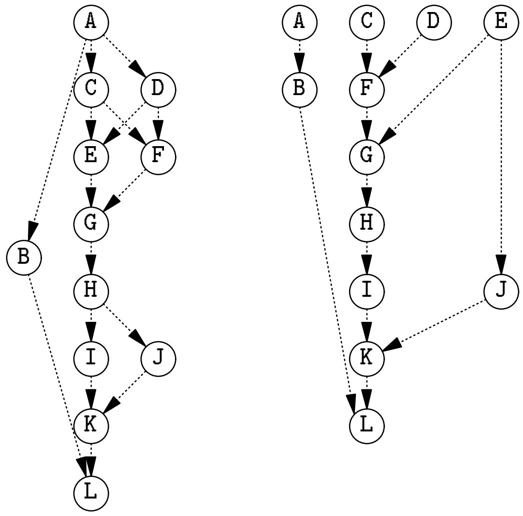
Fig. 2. Comparison of PIR and SPIR with RRD for a 6 issue architecture.

The results show that PIR performs consistently better than SPIR. We were surprised to find that the priority function that considers slack times tended to degrade performance rather than improve performance. Examination of several cases revealed that more spill code was generated because either some values were spilled prior to attempts to sequentialize live ranges, or values were selected that had less of an impact on reducing the size of the excessive requirements. The consideration of slack time tended to negate the effects of considering the number of interferences in the cost function.

PIR and SPIR identify excessive sets using a simplified interference graph. The excessive sets computed by register reuse dags are conceptually similar to those computed by PIR/SPIR. In both cases the sets represent the instructions that the respective heuristics believe will interfere and cause excessive demands.

Thus, it is possible to substitute excessive sets for the simplified interference graphs used by PIR and then proceed using coloring's priority function and spill code generation.

```
A: load a
B: b = 2 * a
C: c = a + 1
D: d = a - 3
E: e = c * d
F: f = c - d
G: g = e / f
H: h = g + 5
I: i = h * 2
J: j = h + 4
K: k = i / j
L: l = b + k
```



(a) 3 address code

(b) Data Dependence DAG

(c) Register Reuse DAG

Fig. 3. Example code and corresponding Reuse DAGs

In Figure 3(a) we show the 3 address code for a basic block, the data dependence dag using statement labels for the code in Figure 3(b), and the register reuse dag in Figure 3(c). The register reuse dag is used to determine the excessive sets of registers. The reuse dag chains those values that are not simultaneously live and can thus share a register under all schedules allowed for parallel execution. In the example, the values computed by statements A, C, D and E can all be alive at the same time and thus cannot share registers. Likewise, the values computed by B G and J cannot share registers since they are on separate chains. If the architecture does not have 4 registers, then the set A, C, D, E is an excessive set.

To compare the performance of excessive sets with the PDG-interference graph we implemented the above modification resulting in the register reuse dag (RRD) algorithm. The comparison of RRD and PIR determines the benefit of excessive sets over interference graphs. The results that were obtained show that the amount of spill code generated in RRD was significantly less than the amount generated in PIR (see Figure 2). We examined numerous cases to verify the results and found a common occurrence mentioned in Briggs' dissertation [5]. Although all nodes in the reduced interference graph interfere with at least

K other values, those K values may not need all K colors. Excessive set measurement computations realize when such a situation occurs and count fewer interferences. The result of fewer interferences is that either a smaller excessive interference set is generated in comparison to the interference graph reduction, or no excessive interference set is generated while interference graph reduction does generate one. The better performance of RRD in comparison to PIR is directly due to this effect.

By comparing the results of PIR with the results in the previous section we observed that although PIR sometimes performs marginally better than IPS, in many situations IPS performs significantly better than PIR. Furthermore, PIR consistently performs significantly worse than ILS. This is because ILS makes use of live range splitting while PIR does not. In summary our experimentation shows that the overall performance of the on-the-fly approach is better than the interference graph approach. We also note that the difference in the performance of various algorithms is greater for higher register pressures. Once enough registers are available, the difference in the performances of the various algorithms is relatively small indicating that integration becomes less important.

5 Unified Resource Allocation using Reuse Dags and Splitting

Finally we present an algorithm that uses register reuse dags and live range splitting. This algorithm is based upon an approach that provides a uniform view of instruction scheduling and register allocation by treating both of them as resource allocation problems. Instruction scheduling is viewed as the allocation of functional units in this approach. Integration is achieved by simultaneously allocating both functional unit and register resources to an instruction. Due to its unified treatment of resources, this approach is referred to as the unified resource allocation approach or URSA [1, 3, 2, 4]. Algorithms that use the URSA approach are based upon the *measure-and-reduce* paradigm. In this approach the areas of the program with excessive resource requirements are located and reductions are performed by transforming the intermediate representation of the program. The selection of a reduction is based upon its effect on the critical path length.

The URSA framework provides a set of techniques to compute resource requirements. When compiling a program to exploit ILP, the dependencies in an acyclic segment of the program are used to represent the set of all semantically correct ways of scheduling the segment. Different schedules may result in different resource requirements. The approach taken in the measure-and-reduce paradigm is to remove from consideration all schedules that result in excessive resource demands using the reduction techniques. Any schedule selected from the remaining set of schedules is feasible with respect to the available resources. Thus, the measurement technique must consider the worst case schedule for each resource to compute the *maximum* resource requirements.

In addition to computing the maximum number of resources required, the measurement techniques must identify the locations in a program where there are excessive resource demands and the locations where a resource is underutilized and available for additional allocations. The areas of overutilization are referred to *excessive sets* and the areas of underutilization are called *resource holes*. The GURRR intermediate representation has been developed to explicitly incorporate maximum resource requirements, excessive sets and resource holes [2]. This representation used in URSA combines information about a program's requirements for both registers and functional units with scheduling information in a single DAG-based representation. In this manner, GURRR facilitates the determination of the impact of all scheduling and allocation decisions on the critical path length of the code affected. GURRR extends the instruction level PDG by the addition of resource hole nodes and reuse edges, which connect nodes that can temporally share an instance of a resource.

The URSA framework supports a set of techniques to perform the allocation of resources to instructions. The techniques utilize the resource holes in GURRR during this process. These techniques are referred to as *resource spackling* techniques because they perform allocations by trying to fill the resource holes with instructions [3]. Register holes represent the cases where a register can be assigned to hold a value. There are two such cases: when the register is unoccupied and when the register is occupied but the value in it is not currently being referenced, and so, the live range can be split. The spackling of an instruction may require live range splitting corresponding to one value in the former type of hole and of two values in the latter.

Instructions belonging to excessive sets are spackled to eliminate excessive resource requirements. The selection of nodes for spackling from excessive sets is based upon priority functions. We considered two different priority functions: the first, URSA-1, selects nodes with the most amount of slack first while the second, URSA-2, selects the same nodes as URSA-1 but spackles them in reverse order. Since instructions with greater slack time have a higher flexibility in their scheduling times, we expected the second priority function to perform better. Different options for spackling an instruction are evaluated by comparing the increases in estimated execution times that are expected as a result. This estimate is obtained as the product of increase in critical path and the execution count of the control dependence region under consideration.

First let us compare the performance of the URSA algorithms with ILS and IPS. As indicated by the results in Figure 4, URSA performs better than ILS on our architecture. Since the difference in the performance of URSA and ILS was lower than we expected, we decided to further investigate the behavior of URSA's reduction techniques. We examined the code generated by URSA for some of the benchmarks to determine if further improvements could be achieved. We found that through *handcoding* we were able to reduce the amount of spill code significantly. As shown in Table 1, the critical path length (CPL) and the number of instructions for loop2 and loop10 can be greatly used through handcoding.

Further examination of the above benchmarks revealed the reason for URSA's

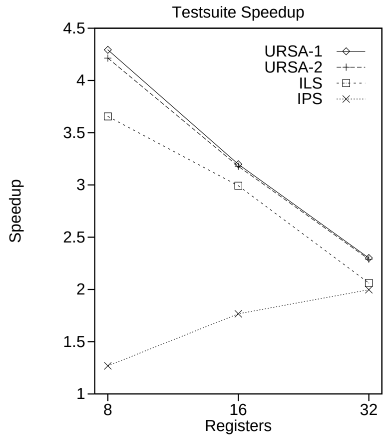


Fig. 4. Comparison of URSA with ILS and IPS for a 6 issue architecture.

Benchmark	CPL URSA-2	CPL Handcoding	Insts. URSA-2	Insts. Handcoding
loop2	95	22	79	40
loop10	181	42	150	71

Table 1. URSA-2 vs Handcoding

inability to discover solutions with less spill code. In situations with high degree of ILP, it is beneficial to apply sequentialization reduction to *groups* of related instructions rather than applying reductions to *individual* instructions. In particular, if the dependence graph contains two relatively independent subdags which are parts of excessive sets, sequentialization of the entire subdags with respect to each other can greatly lower resource requirements without introducing spill code. On the other hand when URSA selected instructions for sequentialization one at a time, it tended to interleave the execution of the two subdags thus requiring spill code. The above observation clearly indicates that URSA's reductions can be enhanced to further improve performance. However, the same cannot be said for ILS since list scheduling uses a greedy approach rather than the measure-and-reduce paradigm.

We also found that the priority function which considers instructions with most slack time first (i.e., URSA-1) does consistently better than the one that considers instructions with least slack time first (i.e., URSA-2). This result was a bit unexpected due to the fact that Goodman and Hsu [11] recommend the scheduling of instructions with the least amount of slack time first. This experiment suggests that scheduling the instructions with the most slack first achieves better performance because these instructions are most likely to be moved beyond the range of the excessive set. Thus fewer reduction transformations are typically required.

The performance of RRD is worse in comparison to URSA because RRD uses live range spilling while URSA employs live range splitting (see Figure 5). Again the difference in the performance of various algorithms diminishes as larger number of registers are made available.

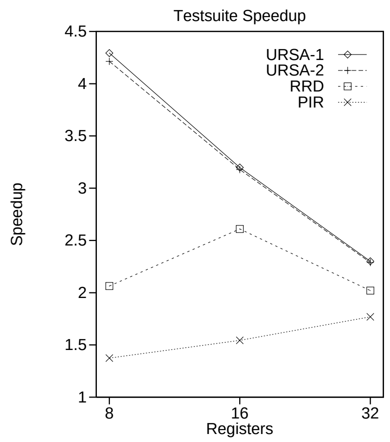


Fig. 5. Comparison of URSA with RRD and PIR for a 6 issue architecture.

Finally we compared the compile-time costs of the various approaches. As shown by the results in Figure 6, URSA based algorithms required the lowest compilation times and the ILS algorithm was the next best performer. The IPS algorithm is slower because it required a register coloring phase following the scheduling to carry out spilling. Finally, RRD ran slower than PIR due to the excessive set computations required by RRD. The interesting aspect of the results is that the algorithms that generated higher quality code also exhibited lower compile-time costs. It should be noted that heuristics implemented were prototypes and there are known areas of improvement for each.

6 Conclusion

In this paper, we presented various versions of algorithms that implement the integration of register allocation and instruction scheduling. From the experiments we conclude that URSA has the overall best performance in integrating register allocation and instruction scheduling. URSA exploits excessive sets that are more accurate than interference graphs in determining the excessive register demands. It also uses live range splitting that performs better than live range spilling. Furthermore, URSA is also efficient in terms of its compilation time costs.

Our results indicate that the on-the-fly register allocation scheme when used with live range splitting always performed better than the interference graph

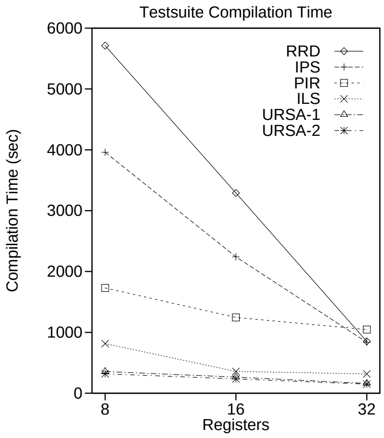


Fig. 6. Compile-time costs of various approaches.

approach. When only considering on-the-fly register allocation with scheduling technique, we show that the ILS technique proposed by us that uses live range splitting performs much better than Goodman and Hsu's IPS technique [11] that uses live range spilling.

Finally a general trend was observed in all the experiments. The difference in the performances of different heuristics grew smaller as greater numbers of registers were made available. This is because the higher the register pressure the greater is need for effective integration of register allocation and instruction scheduling.

Results of two additional studies that have considered the interaction between instruction scheduling and register allocation were reported by Bradlee et al. [6] and Norris et al. [13]. In contrast to these studies, our study shows a greater degree of variation in the performance of different algorithms and thus indicating a greater significance of the impact of integration on performance. We believe this is due to the fact that our study is the only one that consider an architecture with a high degree of ILP. Both of the earlier studies were performed in context of single issue pipelined machines capable of exploiting only low degrees of ILP.

References

1. David A. Berson, Unification of register allocation and instruction scheduling in compilers for fine grain architectures. *Ph.D. Thesis, Dept. of Computer Science, University of Pittsburgh, Pittsburgh, PA*, November 1996.
2. David A. Berson, Rajiv Gupta, and Mary Lou Soffa. GURRR: A global unified resource requirements representation. In *Proc. of ACM Workshop on Intermediate Representations, Sigplan Notices*, vol. 30, pages 23–34, April 1995.
3. 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.

4. 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 Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254, 1993.
5. Preston Briggs. Register allocation via graph coloring. *Ph.D. Thesis, Dept. of Computer Science, Rice University, Houston, TX*, April 1992.
6. David Bradlee, Susan Eggers, and Robert Henry. Integrating register allocation and instruction scheduling for riscs. In *Proceedings of ASPLOS*, April 1991.
7. Claude-Nicholas Fiechter, PDG C Compiler. Technical Report, Dept. of Computer Science, University of Pittsburgh, Pittsburgh, PA, 1993.
8. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–58, 1981.
9. F. Chow and J. Hennessy. Register allocation by priority-based coloring. *ACM Trans. Prog. Lang. and Systems*, 12(4):501–536, 1990.
10. 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.
11. 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.
12. 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.
13. Cindy Norris and Lori L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. *Proceedings of MICRO-28*, Nov. 1995.
14. Cindy Norris and Lori L. Pollock. A scheduler-sensitive global register allocator. *Proceedings of Supercomputing'93*, pages 804-813, Portland, Oregon, 1993.
15. 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.