Low-cost Register-pressure Prediction for Scalar Replacement Using Pseudo-schedules*

Yin Ma
Steve Carr
Michigan Technological University
Department of Computer Science
Houghton MI 49931-1295
{*yinma,carr*}*@mtu.edu*

Rong Ge
Simon Fraser University
School of Computing Science
Burnaby BC Canada V5A 1S6
rge@cs.sfu.ca

Abstract

Scalar replacement is an effective optimization for removing memory accesses. However, exposing all possible array reuse with scalars may cause a significant increase in register pressure, resulting in register spilling and performance degradation. In this paper, we present a low cost method to predict the register pressure of a loop before applying scalar replacement on high-level source code, called Pseudo-schedule Register Prediction (PRP), that takes into account the effects of both software pipelining and register allocation. PRP attempts to eliminate the possibility of degradation from scalar replacement due to register spilling while providing opportunities for a good speedup.

PRP uses three approximation algorithms: one for constructing a data dependence graph, one for computing the recurrence constraints of a software pipelined loop, and one for building a pseudo-schedule. Our experiments show that PRP predicts the floating-point register pressure within 2 registers and the integer register pressure within 2.7 registers on average with a time complexity of $O(n^2)$ in practice. PRP achieves similar performance to the best previous approach, having $O(n^3)$ complexity, with less than one-fourth of the compilation time on our test suite.

1. Introduction

Traditional optimizing compilers have effective global register allocation algorithms for scalars. Now, many commercial compilers extend register allocation by replacing multiple array references to the same memory location with references to scalars to allow global register allocation to put these values in registers. This replacement of arrays with scalar temporaries is called *scalar replacement* [2, 3, 4, 5, 6, 8].

Because traditional scalar register allocation algorithms cannot allocate array elements to registers, scalar replacement becomes a bridge to expose the opportunities for allocating array elements to registers using traditional algorithms. Consequently, scalar replacement reduces the number of memory accesses and improves performance. Unfortunately, converting all possible array reuse to scalars may dramatically increase the register pressure in short fragments of code, resulting in performance degradation in loops due to increased register spilling [6]. Aggressive scheduling techniques such as software pipelining are sensitive to register demands, magnifying the change in register pressure from scalar replacement. If the extent of scalar replacement can be controlled precisely, those risks are likely to be eliminated and scalar replacement may always provide a positive speedup. To achieve this goal, we present a new algorithm, called *Pseudo-schedule Register Prediction* (PRP), that predicts the register pressure of a given loop before scalar replacement and software pipelining are applied. PRP achieves good performance with $O(n^2)$ complexity in practice.

PRP determines register pressure directly from the source code by estimating the effects of optimization and software pipelining without actually performing optimization and complete software pipelining. Performing scalar replacement on unoptimized source allows the compiler to retain the structure of the original code, making it easier to gather loop and array information. By avoiding optimization and full software pipelining, PRP becomes significantly cheaper since the code is not fully compiled both to predict pressure and to generate the final code. While these simplifications lower the accuracy of PRP, we show that on average PRP produces register pressure estimations usually within 0, 1 or 2 registers of the actual pressure.

*This research has been partially supported by NSF grant CCR-0209036.

We begin this paper with a brief discussion of previous work on register-pressure prediction. Then, we give a review of scalar replacement and software pipelining. Next, we present PRP and our experimental results. Finally, we give our conclusions and discuss future work.

2. Previous Work

Wolf, et al. [13], present a technique for ensuring that unroll-and-jam, scalar replacement and software pipelining do not use too many registers. They consider the effects of the pipeline filling requirements and the number of scalars needed for scalar replacement when modeling the floating-point register requirement before loop transformations. Their method defines pipeline filling requirements of a processor as the number of floating-point registers required to keep the pipeline running at full speed. To get the total register pressure of a loop, the number of scalars needed for scalar replacement is added to the pipeline filling requirements. This technique will overestimate the number of registers required since the registers reserved for pipeline filling may not all be needed.

Carr, et al. [3, 5], estimate register pressure before applying scalar replacement by reserving an experimentally determined number of registers for scheduling and adding that amount to the number of scalars used in a loop by scalar replacement. As in Wolf's method, this technique may reserve more registers than necessary to allow for the increased register pressure due to scheduling.

Huff [10] presents a scheduling method to apply software pipelining with minimum register pressure. Huff defines, two metrics, *MaxLive* and *MinAvg*, to measure the register pressure. MaxLive represents the lower bound on the number of registers required by a software pipelined loop. It is very accurate but available only after a software pipeline has been created, making it too costly and too late for prediction. MinAvg also represents a lower bound on register pressure, but its computation occurs before pipelining. MinAvg is defined as the sum of minimum lifetimes of all variables divided by the initiation interval (II), where II is the number of cycles between the initiations of two consecutive iterations in a loop. Unfortunately, MinAvg ignores the effects of overlapping lifetimes. In addition, MinAvg assumes all variables can be scheduled such that minimum lifetimes are achieved. Under high resource constraints minimum lifetimes are often not achieved, resulting in a highly inaccurate register pressure estimation.

Ding [7] proposes a different approach that uses MinDist to compute register pressure. MinDist is a two dimensional array used to compute software pipelining dependence constraints that contains information about the minimum lifetime of each variable. Ding claims that the overlapping in software pipelining requires additional registers only when the lifetime of a variable is longer than II. Since MinDist gives the lower bound of the lifetime of a variable, the number of registers for this variable can be directly predicted as $\left\lceil \frac{Lifetime}{II} \right\rceil$. MinDist, however, ignores resource constraints, resulting in an imprecise prediction under high resource constraints.

Recently, Ge [9] described a new method that uses the information in MinDist to build a schedule as an approximation of the real schedule for predicting register pressure. DU chains are computed based on the approximate schedule. The maximum number of DU chains overlapped in a cycle will be the number of registers predicted. By her observation, long DU chains and aggregated short DU chains reflect the effect of high resource conflicts. She presents two heuristic algorithms to handle these two types of chains.

Ge's method predicts register pressure more accurately than MinAvg and MinDist methods at the intermediate language level. However, our objective is to predict register pressure at the source level before applying scalar replacement. Ge's method can give imprecise prediction on source code because it relies on a complete DDG. Moreover, computing MinDist and the length of recurrences takes $O(n^3)$ time. This cost is higher than desired for our purposes.

3. Scalar Replacement and Software Pipelining

In this section, we review scalar replacement and software pipelining, motivating the need to predict register pressure to limit potential performance degradation in the presence of high register pressure.

3.1. Scalar Replacement

Scalar replacement is a loop transformation that uses scalars, later allocated to registers, to replace array references to decrease the number of memory references in loops. In the code shown below, there are three memory references and one floating-point addition during an iteration.

```
for ( i = 2; i < n; i++ )
  a[i] = a[i-1] + b[i];
```

The value defined by `a[i]` is used one iteration later by `a[i-1]`. Using scalar replacement to expose this reuse, the resulting code becomes

```
T = a[1];
for ( i = 2; i < n; i++) {
  T = T + b[i];
  a[i] = T;
}
```

Here the number of memory references decreases to two with the number of arithmetic operations remaining the same. If the original loop is bound by memory accesses,

scalar replacement improves performance. On the other hand, more scalars are used, demanding more registers to hold their values. The resulting increased register pressure may cause excessive register spilling and degrade performance [6].

3.2. Software Pipelining

Software pipelining [1, 11, 12] is an advanced scheduling technique for modern processors. One popular method for software pipelining, modulo scheduling [12], tries to create a schedule with the minimum number cycles used for one iteration of a loop such that no resource and dependence constraints are violated when this schedule is repeated.

The *initiation interval* (II) of a loop is the number of cycles between the initiation of two consecutive iterations. The initiation interval gives the number of cycles needed to execute a single iteration of a loop and determines the loop's performance. Given a loop and a target architecture, compilers can predict the lower-bound on II. The resource initiation interval (ResII) gives the minimum number of cycles needed to execute the loop based upon machine resources such as the number of functional units. The recurrence initiation interval (RecII) gives the minimum number of cycles needed for a single iteration of the loop based upon the length of the cycles in the DDG. The maximum value between RecII and ResII, called the minimum initiation interval (MinII), represents a lower bound on the minimum value of an II.

Software pipelining can significantly improve loop performance. Consider a loop L that iterates n times and contains three instructions we call A, B, and C. Assume that dependences in the loop require a sequential ordering of these operations within a single loop iteration. Thus, even if our target architecture allows 3 operations to be issued at once, a schedule for a single loop iteration would require 3 instructions due to dependences among the operations. The resulting loop would execute in $3 * n$ cycles on a machine with one-cycle operations.

A software pipelined version of L might well be able to issue all three operations in one instruction by overlapping execution from different loop iterations. This might under ideal circumstances, lead to a single-instruction loop body of $A^{i+2}B^{i+1}C^{i}$ where X^{j} denotes operation X from iteration j of the loop [1]. The cost of the software pipelined loop is about one-third of the cost of the original loop, namely $n+2$ cycles including the prelude and postlude. So software pipelining can, by exploiting inter-iteration concurrency, dramatically reduce the execution time required for a loop.

Unfortunately, overlapping of loop iterations also leads to additional register requirements. For illustrative purposes, assume that operation A computes a value, v, in a register and that operation C uses v. In the initial sequential version of a loop body one register is sufficient to store v's value. In the software pipelined version, we need to maintain as many as three different copies of v because we have different loop iterations in execution simultaneously. In this particular case, the register holding the value from iteration i can be used by operation C and defined by operation A in the same instruction since reads from registers occur before writes to registers in the pipeline. Thus, we would need 2 registers for v.

Since software pipelining and scalar replacement can demand a larger number of registers, we must predict register pressure before applying them, or risk degrading performance. In the next section, we detail our register-pressure prediction algorithm.

4. Pseudo-schedule Register Prediction (PRP)

To predict the number of registers used by software pipelining directly from high-level source code, we must build a sufficiently detailed DDG to represent the loop body, approximate the RecII and approximate the final software pipeline. This section describes our approach, PRP, that attempts to balance the detail needed for an accurate prediction with the desire to make the prediction as efficient as possible.

4.1. Constructing the DDG

In this section, we present a low-cost algorithm to generate a sufficiently precise DDG directly from high-level source code. This algorithm only applies to innermost loops that contain only assignment statements with array references, scalars and arithmetic operators. The construction algorithm presented in this section is based on an *abstract syntax tree (AST)*. In addition to an AST, the algorithm utilizes the array data dependence graph to predict the effects of scalar replacement.

Nodes in the DDG represent intermediate operations and edges between nodes represent data dependences. The label on an edge is a vector $\langle$Delay,Diff$\rangle$, where Delay represents the minimum number of cycles that must occur between the two operations incident on this edge and Diff represents the number of loop iterations between the two adjacent nodes.

Before construction, we prune the original array dependence graph to represent the flow of values between array references as is done in scalar replacement [3]. After analyzing the pruned graph, we build a reuse map, called *reuseMap*, from each array reference to one of three states: *generator*, *replaced*, and *move-out-of-loop*. *Generator* marks an array reference that will be kept after scalar replacement. *Replaced* indicates references eliminated during scalar replacement, and *move-out-of loop* indicates those references that are loop invariant. Any array reference marked as *replaced* or *move-out-of-loop* will not be put in

the DDG. Besides array references, reuse may also happen in index addressing and scalars. In our algorithm, reuse maps for index addressing and scalars are updated during DDG generation.

The approximate DDG construction algorithm must remain consistent with the expected output of the optimizer. We have found that optimization results are quite predictable for loops amenable to scalar replacement. While the DDG construction is tailored to the optimizer we are using, we expect that the concepts are more widely applicable since the optimization passes are standard and generally predictable. The algorithm reads each element in an AST in execution order and generates the corresponding DDG nodes. Any specialized instruction generation should be reflected in the DDG generation. Because constants and scalars defined outside loops don't need new registers, no DDG nodes are generated for them. For array references, scalars and index subscripts, we always use old nodes if reuse is available; otherwise, we add this node to *reuseMap*. Edges are added if two nodes have a dependence. The Delay and Diff on edges are determined by the targeted architecture. Loads and stores have dependences if they may access the same memory location. To represent the dependences in the AST and model the backend scheduler that we use, we assume that a store has dependences with all loads generated in the same statement and a load has a dependence only with the latest generated store.

The key to generating a good DDG for prediction is to consider reuse thoroughly. Each reuse critically impacts the shape of a DDG because reused operations share a set of nodes. This, in turn, affects the prediction of register pressure. Without a precise DDG, PRP produces an inaccurate register-pressure prediction. Figure 2A gives an example DDG constructed by our algorithm.

4.2. RecII Approximation

After generating a DDG, PRP generates a pseudo-schedule used for register-pressure prediction. Scheduling fills out a table where each column corresponds to one available functional unit in the target architecture, and each row represents an execution cycle. The number of rows in the table is the MinII, the maximum value of RecII and ResII. An accurate register pressure prediction requires a precise estimation of MinII. Unfortunately, traditional methods for computing RecII, such as using MinDist, are $O(n^3)$ where n is the number of nodes in a DDG. This is too expensive to use in prediction. To limit the cost of PRP, we present a heuristic algorithm that is close to the $O(n^2)$ in most situations.

In practice, a DDG generated by a compiler is not an arbitrary graph but one with a certain topological order already. If we give each node a sequence number in the order of being generated, the node numbers obey the order of execution. So we define a forward edge as an edge from a node with a smaller sequence number to one with a larger number and a backward edge as an edge from a source node sequence number greater than or equal to the sink node sequence number. With this order, our algorithm approximately computes RecII in three steps.

The first step converts a DDG from a directed graph to an undirected graph called the *SimpleDDG*, in which there is only one edge between each pair of nodes. This is shown in Figure 1A. Then, for each pair of nodes, we merge the vectors of one forward edge with the largest Delay and one backward edge with the smallest Diff together to be forward and backward vectors of a new undirected edge, as shown in Figure 1A. In Figure 1A, the first vector is the forward vector and the second vector is the backward vector. If there are only forward edges or backward edges, we leave the corresponding vector blank or give special values, such as [1,0][-10000,-10000], where -10000 means there is no edge between the two nodes.

After creating the undirected graph, in the second step we simplify it using four transformations. The algorithm visits each node and applies transformations repeatedly until no more simplifying is possible. A detailed description of the four transformations is given below.

Transformation 1: If there is an edge having the same source and sink, compute the potential RecII as if both the forward vector and backward vector have valid values using the equation below.

$$\text{Potential RecII } = \frac{D_f + D_b}{F_f + F_b}$$

where D_f is delay in the forward vector, D_b is delay in the backward vector, F_f is diff in the forward vector and F_b is diff in the backward vector. Then, remove this edge. See Figure 1B for an example where the potential RecII is 3 by applying the equation $\frac{2+1}{0+1}$.

Transformation 2: If a node only has one edge connecting it to other nodes, then remove this node and the edge with it, as shown in the Figure 1C.

Transformation 3: If a node has degree two, merge its two edges by adding the values in the matching elements in their vectors, as shown in the Figure 1D.

Transformation 4: If there are two edges between a pair of nodes, merge the two edges. The forward vectors with the largest Delay/Diff ratio will be the forward vector of the new edge. The same is true of the backward vector. Two potential RecIIs are computed if a cycle can go clockwise and counterclockwise. For the clockwise cycle, we use the forward vector of the first edge

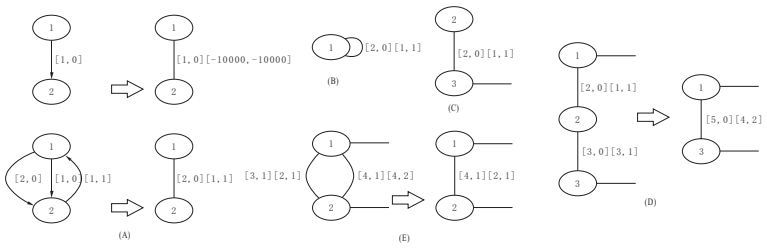

Figure 1. DDG Simplification Transformations

and the backward vector of the second edge to compute the RecII. For the counterclockwise cycle, we use the backward vector of the first edge and the backward vector of the second edge to compute another RecII, as shown in the Figure 1E. In the example, one potential RecII is $\frac{7}{3}$ computed by $\frac{3+4}{1+2}$ and another potential RecII is 3 computed by $\frac{4+2}{1+1}$.

After graph simplification, the graph may be empty. In this case, we take the maximum potential RecII computed during simplification as the approximate RecII. However, if nodes remain in the graph, such as the simplification from Figure 2A to Figure 2B, we apply a third step using the algorithm in Figure 3 to compute one more potential RecII on the simplified graph. The maximum of all computed potential RecIIs is our approximate RecII.

4.3. Register-pressure Prediction

PRP uses the standard kernel scheduling algorithm to fill out a schedule table but ignores all backward edges in the *SimpleDDG*[12]. The ResII is computed from the approximate DDG and the target architecture. Once the compiler generates a pseudo-schedule, it computes the lifetime for each potential virtual register. In the *SimpleDDG*, the definition of a virtual register is implicit because multiple nodes may map to one expression in the source, indicating they share one virtual register. If a node has at least one forward edge, this node represents the definition of a virtual register. The last use of this virtual register is the latest-generated sink node on the forward edge with the largest lifetime (LT), where LT $=$ Diff $*$ II $+$ Delay. For instance, if an edge has the vector $\langle 3, 1\rangle$ and the II is 9, its LT will be $9 * 1 + 3 = 11$ cycles.

Def	1(0)	2(0)	3(0)	5(1)	7(1)	9(2)	11(0)
Use	2(0)	9(2)	4(1)	6(1)	8(2)	12(2)	12(2)

Table 1. The Define-Use List for Figure 2A

To compute LT for each virtual register, we consider the DU-chains in the DDG. Consider the def-use table for Figure 2 in Table 1. Each column in the table is a def-use pair. The first number is the node number and the second one is the cycle in which this node is scheduled. With the Def-Use table and the schedule table, we will use the DU-Chain technique introduced by Ge to predict the final register pressure [9]. The idea behind this technique is if there are two def-use pairs, $A \rightarrow B$ and $B \rightarrow C$, we consider the chain $A \rightarrow B \rightarrow C$. If there is a branch such as $A \rightarrow B$ with $B \rightarrow C$ and $B \rightarrow D$, we choose the chain $A \rightarrow B \rightarrow C$ instead of $A \rightarrow B \rightarrow D$ if $B \rightarrow C$ has a longer LT or C is generated later than D if both LTs are equal. So the integer DU chains for Figure 2A are $1 \rightarrow 2 \rightarrow 11$, $3 \rightarrow 4 \rightarrow 10$, $6 \rightarrow 7$, $8 \rightarrow 9$, and $12 \rightarrow 13$. After drawing all chains into the schedule table, we apply Ge's adjustment methods on long DU chains. A long chain is defined as any chain whose length is greater than or equal to the II. A node may be delayed in real schedules for resource conflicts or backward edges, causing inaccuracies in the pseudo-schedule. A delay in one node of a chain will result in the delay of successor nodes. The longer the chain, the more likely its last node may be delayed. To handle long chains, we increase the iteration number of the last node of the chain by one. Our experiments show that the register-pressure prediction after this adjustment is closer to the actual register pressure. Finally, the predicted register pressure is the maximum number of DU chains passing through a cycle after adjustments.

4.4. The Effects of Loop Unrolling

Some scalar replacement algorithms use unrolling to eliminate register-to-register copies. The unroll factor of scalar replacement is determined by the maximum distance of a dependence edge leaving a generator in the DDG. Our experiments have shown there is only a small change in the number of floating-point registers used after unrolling. The integer register pressure is strongly correlated with the unroll factor. In our method, we use the equation below to handle the effects of unrolling:

$$I_a = I_b * \left(1 + \frac{2}{3} * \text{unroll factor}\right)$$

where I_a is the integer pressure after unrolling and I_b is the integer pressure before unrolling. For floating-point register pressure prediction, we make no adjustments.

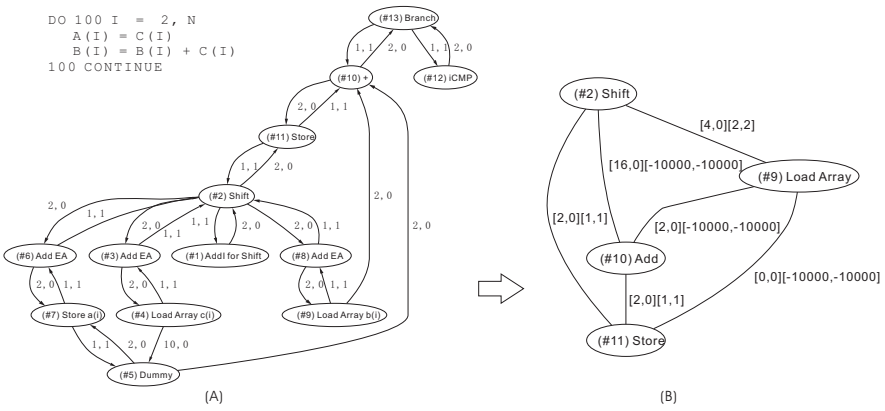

Figure 2. Example DDG Simplification

4.5. The Effects of Software Pipelining

Software pipelining aims at fully utilizing the idle functional units. It schedules using information in a DDG, including backward edges, that is ignored by our method. When the RecII is larger than the ResII, the algorithm will underestimate the register pressure. To compensate for the underestimation, we update the prediction using the following equations:

$$I_a = \lceil I_b + c \times (T - I_b) \rceil$$
$$F_a = \lceil F_b + c \times (T - F_b) \rceil$$

where

$$\begin{array}{ll} c = 0 & \text{if } \frac{RecII}{ResII} < 1.5 \\ c = 0.3 & \text{if } 1.5 \leq \frac{RecII}{ResII} < 2 \\ c = 0.6 & \text{if } \frac{RecII}{ResII} \geq 2 \end{array}$$

The values of c come from observation. F_a and I_a are the floating-point and integer register pressure after adjustment, respectively. F_b and I_b are the floating-point and integer pressure before adjustment, respectively. T is the total number of registers used. The larger the difference between RecII and ResII, the greater the impact on register pressure by software pipelining. The coefficient c may have other values for other software pipelining algorithms.

5. Experiment

We have implemented PRP in Memoria, a source-to-source Fortran transformer. Memoria performs scalar replacement, unroll-and-jam and a number of other loop transformations. After scalar replacement, we use Rocket to generate the software pipeline using iterative modulo scheduling [12] and determine the register pressure of the final loop. The machine-independent optimization passes used include constant propagation, global value numbering, partial redundancy elimination, strength reduction and dead code elimination.

For our experiments, we have chosen a target architecture that has two integer functional units and two floating-point functional units. Both integer and floating-point instructions have a latency of two cycles. In this experiment, we investigate 162 loops extracted from the SPEC95 benchmark suite on which both scalar replacement and iterative modulo scheduling are applicable. Each loop is saved into an individual file. The register pressure of a loop is the sum of registers defined in a loop body and live-in registers.

We compare the performance of PRP to the method described in Ge's thesis [9]. Our implementation of Ge's method uses our DDG generation algorithm in Memoria. Ge's method has been shown to be superior to the MinAvg approach developed by Huff [10] and the MinDist technique proposed by Ding [7]. Ge reports that MinAvg and MinDist give an average register mis-prediction of 4–5 registers on an exact low-level DDG. PRP achieves much better results on an approximate DDG as reported in the next section.

5.1. Accuracy of Register-pressure Prediction

Table 2 reports the accuracy of our register-pressure prediction algorithm. The column "Avg Err" reports the average of the absolute value of the difference between the predicted register pressure and the actual register pressure obtained by Rocket. "Rel Err" gives the average relative error of the prediction.

PRP gives the highest prediction accuracy. This algorithm predicts the floating-point register pressure within 1.99 registers and the integer register pressure within 2.65 registers, on average. Ge's method predicts floating-point pressure within 2.53 registers and integer register pressure within 3.32 registers on average. Theoretically, Ge's method should be better than PRP, or at least equivalent to it since the DDGs are the same and Ge's method does a better job of scheduling and predicting RecII. The problem is that in Memoria we do not have enough information in the approximate DDG to compute Ge's method accurately. For example, information regarding outer-loop induction vari-

Benchmark	#Loops	PRP				Ge			
		Integer		Floating-Point		Integer		Floating-Point	
		Avg Err	Rel Err	Avg Err	Rel Err	Avg Err	Rel Err	Avg Err	Rel Err
101.tomcatv	4	0.50	0.04	1.50	0.23	1.50	0.10	1.00	0.10
102.swim	10	2.80	0.13	2.50	0.27	3.60	0.16	3.40	0.28
103.su2cor	36	3.44	0.21	1.44	0.25	3.81	0.22	3.06	0.36
104.hydro2d	55	1.82	0.16	1.67	0.38	2.36	0.19	1.84	0.37
107.mgrid	2	3.00	0.50	1.00	0.25	2.00	0.33	1.00	0.25
110.applu	31	2.52	0.18	2.35	0.22	3.90	0.22	3.16	0.27
125.turb3d	16	3.38	0.18	3.00	0.32	5.69	0.26	2.88	0.33
141.apsi	8	4.63	0.39	3.00	0.84	1.63	0.12	1.88	0.67
All Loops	162	2.65	0.19	1.99	0.20	3.32	0.33	2.53	0.35

Table 2. Prediction Accuracy

ables is not present in the approximate DDG. Ge's method relies heavily on an exact DDG and, thus, achieves accuracy less than PRP when predicting register pressure on an approximate DDG.

PRP performs better on applications such as 101.tomcatv and 104.hydro2d than on applications such as 125.turb3d and 141.apsi. The reason has to do with the complexity of the DDG. 125.turb3d and 141.apsi have a higher percentage of loops with a more complex DDG, increasing the errors in the total effect of the approximation algorithms. Benchmarks like 101.tomcatv and 104.hydro2d have less complex DDGs and experience less error.

To compare the savings in compilation time due to the approximate RecII computation (the most computationally expensive portion of prediction), we evaluate the total compilation time of PRP using the RecII approximation algorithm and the total compilation time of PRP using MinDist to compute the RecII on a 2.4GHz AMD Athlon XP. PRP with the RecII approximation algorithm requires $389ms$ of compilation time over the set of benchmark loops while PRP with MinDist requires more than four times the compilation time, $1808ms$.

PRP is approximate. Impreciseness in DDG construction, RecII computation and scheduling brings errors into the final results. Generally, the prediction is worse than average in loops with unrolling in scalar replacement and loops with their RecII much larger than their ResII because we use equations as a simple model of their effects. To make a detailed analysis on the process of error accumulation, we tested the accuracy of each step separately below.

5.2. Accuracy of DDG Construction

A precise DDG provides an essential foundation for good register-pressure prediction. To measure the preciseness of our approximate DDG, we recorded the absolute value of the difference between the number of integer/floating-point virtual registers predicted from that DDG and the number of integer/floating-point virtual registers used in the low-level intermediate after optimization. We predict the number of integer virtual registers within an average of 1.35 registers and within an average of 0.65 register for floating-point virtual registers. Table 3 give the distribution of the error. The absolute error is computed by taking the absolute value of the difference between the number of virtual registers required by the approximate DDG and the number of virtual registers found in the actual intermediate code after optimization. The entries for each benchmark indicate the number of loops whose absolute error matches the column heading.

Benchmark	Absolute Error						
	0	1	2	3	4	5	>5
	#Loops(Int)						
101.tomcatv	4	0	0	0	0	0	0
102.swim	4	1	3	0	1	0	1
103.su2cor	12	10	7	5	1	1	0
104.hydro2d	35	9	5	0	3	1	2
107.mgrid	0	2	0	0	0	0	0
110.applu	17	3	7	2	1	0	1
125.turb3d	7	1	3	2	0	1	2
141.apsi	1	4	2	1	0	0	0
Total	111	26	16	1	5	0	3
	#Loops(FP)						
101.tomcatv	4	0	0	0	0	0	1
102.swim	8	1	1	0	0	0	0
103.su2cor	24	5	5	0	0	0	2
104.hydro2d	33	13	7	0	2	0	0
107.mgrid	1	0	1	0	0	0	0
110.applu	25	2	2	0	0	0	2
125.turb3d	10	2	1	0	3	0	0
141.apsi	5	1	0	1	1	0	0
Total	79	30	28	9	7	3	6

Table 3. Virtual Register Errors

PRP predicts the number of virtual registers exactly for over half of the test cases. In the rest of them, most of the errors are caused by the failure to identify reuse on scalars because our method only scans innermost loop bodies and doesn't utilize reuse from outside of loops. However, in general, our algorithm can directly and quite precisely construct an approximate DDG graph for loops before scalar replacement and global optimization.

5.3. Accuracy of RecII Approximation

To measure the effectiveness of our RecII approximation, we implemented the algorithm in Rocket and compared the predicted RecII to the actual RecII computed by

```
algorithm Approximation_Process( graph )
    Node n = FindNodeWithMaxDegree( graph )
    int #f = n.theNumberofForwardEdges
    int #b = n.theNumberofBackwardEdges
    Vector F = <0,0>
    Vector B = <0,0>
    if #f >= #b then
        do breadth-first search along all forward edges
            foreach node visited do
                traverse the edge E with the max Delay/Diff ratio
                    in forward vectors
            endforeach
            F = E.forwardVector
        enddo
        do breadth-first search along all forward edges
            foreach node visited do
                traverse the edge E with the min Delay/Diff ratio
                    in backward vectors
            endforeach
            B = E.forwardVector
        enddo
    else
        do breadth-first search along all backward edges
            foreach node visited do
                traverse the edge E with the max Delay/Diff ratio
                    in forward vectors
            endforeach
            F = E.forwardVector
        enddo
        do breadth-first search along all backward edges
            foreach node visited do
                traverse the edge E with the min Delay/Diff ratio
                    in backward vectors
            endforeach
            B = E.forwardVector
        enddo
    endif
    MaxRecII = ComputeRecII( F, B )
```

Figure 3. RecII Approximation

Rocket's software pipelining algorithm. Table 4 gives the results of this experiment. The "Abs Err" row indicates the absolute value of the difference between the predicted RecII and the actual RecII. For 113 out of 162 loops, our algorithm computed the RecII exactly, with an average error of 2.51. For loops with many DDG nodes having a degree more than three, our algorithm usually gives inaccurate results because simplification cannot reduce the complexity of the graphs significantly.

Because simplification uses an iterative algorithm, the number of passes before the algorithm halts becomes a critical factor to measure the efficiency of the entire algorithm. On average, Memoria simplifies the DDG in 2.86 iterations, while Rocket simplifies the graph in an average of 3.59 iterations. Table 5 gives the distribution of the iteration counts. Simplification halts within 5 iterations on 94% of the loops. In practice, we can consider it to be a constant without relation to the number of nodes in graphs. The conversion from a directed graph to an undirected graph only travels each node and read values once for each edge. Therefore, the time complexity of simplification becomes $O(n^2)$, where n is the number of nodes.

	Abs Err				Percentage		Avg Err
Benchmark	0	1	2	>2	<=2	>2	
101.tomcatv	1	0	2	1	75	25	3.00
102.swim	7	0	1	2	80	20	1.60
103.su2cor	21	2	1	12	67	33	3.36
104.hydro2d	50	1	0	4	92	8	0.44
107.mgrid	1	0	0	1	50	50	2.00
110.applu	18	6	0	7	77	23	1.94
125.turb3d	9	0	0	7	56	44	8.63
141.apsi	6	0	0	2	75	25	4.00
All Loops	113	9	4	36	78	22	2.51

Table 4. Accuracy of RecII Approximation

Iterations	2	3	4	5	6	7	8
Rocket	0	87	55	6	3	6	0
Memoria	90	37	21	8	0	1	3

Table 5. Distribution of RecII Iterations

By observation, simplification effectively eliminated the nodes in the DDG and reduced DDG complexity. The result gives us a very good estimation of RecII with a significantly lower cost than computing RecII on the original DDG.

5.4. Accuracy of Scheduling

We tested our scheduling algorithm along with RecII approximation in Rocket. We compute an approximate schedule to predict register pressure from the exact DDG of Rocket before software pipelining. Table 6 details the performance comparison. The columns labeled "Rel Err" report the relative error of the predicted register pressure from the pseudo-schedule compared to the actual register pressure from the actual schedule.

PRP achieves a floating-point register pressure prediction within 1.42 registers and an integer register pressure prediction within 2.03 registers on average. This predictions is better than PRP achieved on the approximate DDG; however, Ge's method achieves the best performance on an exact DDG. With an exact DDG, Ge's method outperforms PRP using the precise RecII and the consideration of backward edges in scheduling. The increased accuracy is a trade-off with speed since Ge's method requires $O(n^3)$ time while PRP runs in close to $O(n^2)$ time.

6. Conclusion

We have presented a method, called Pseudo-schedule Register Prediction (PRP), for predicting the register pressure of loops considering both scalar replacement and software pipelining directly from source code without the need to perform intermediate code optimization or full software pipelining. By performing the prediction directly on source code, we retain high-level information needed in loop optimizations with a significantly lower cost than doing full compilation in order to compute exact register pressure.

	PRP				Ge			
	Integer		Floating-Point		Integer		Floating-Point	
Benchmark	Avg Err	Rel Err	Avg Err	Rel Err	Avg Err	Rel Err	Avg Err	Rel Err
101.tomcatv	1.00	0.07	0.40	0.05	0.80	0.05	0.60	0.08
102.swim	2.90	0.13	1.90	0.18	1.30	0.06	1.00	0.07
103.su2cor	2.75	0.16	1.13	0.14	2.30	0.12	0.83	0.09
104.hydro2d	0.84	0.07	1.05	0.17	0.67	0.05	0.96	0.14
107.mgrid	1.67	0.20	1.00	0.23	0.67	0.03	1.67	0.37
110.applu	1.64	0.11	1.50	0.15	1.56	0.08	1.25	0.10
125.turb3d	3.82	0.18	2.47	0.14	2.76	0.09	2.65	0.11
141.apsi	1.92	0.17	1.17	0.14	1.67	0.17	0.42	0.07
All Loops	2.03	0.15	1.42	0.12	1.52	0.08	1.12	0.11

Table 6. Prediction Accuracy on Exact DDG

PRP includes three approximation algorithms: one for constructing the data dependence graph at the intermediate language level, one for computing RecII and one for building a schedule. The time complexity is $O(n^2)$ in practice compared to the $O(n^3)$ complexity of previous approaches. In addition our experiments show this method has the best performance when predicting register pressure directly from the source code.

We are currently engaged in embedding this prediction algorithm into a standard scalar replacement algorithm and trying to build a model describing the relationship between register pressure and the final speedup scalar replacement can provide. We hope with this model and the prediction algorithm shown in this paper, we can replace only a selected subset of array references to achieve good speedup. We wish to precisely control the register pressure after transformation with our prediction method; consequently, eliminating the possibility of performance degradation caused by excessive replacement.

Only predicting for scalar replacement is not enough because unroll-and-jam is often applied before scalar replacement in order to create more opportunities for reducing the number of memory references and improving parallelism. Unroll-and-jam may increase the demand on registers and cause significant performance degradation. Therefore, we must predict register pressure for unroll-and-jam and scalar replacement accurately.

PRP, as a fast prediction algorithm, is a cornerstone of promising solutions for making the performance of scalar replacement predictable. Without performance degradation, scalar replacement will be not only an efficient optimization but also a feasible and effective one given its low cost and potential high gain.

References

[1] V. Allan, R. Jones, and R.Lee. Software pipelining. *ACM Computing Surveys*, 7(3), 1995.

[2] R. Bodik, R. Gupta, and M. L. Soffa. Load-reuse analysis: Design and evaluation. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 64–76, Atlanta, GA, 1999.

[3] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, NY, 1990.

[4] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, 1988.

[5] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software – Practice and Experience*, 24(1):51–77, 1994.

[6] S. Carr and P. Sweany. An experimental evaluation of scalar replacement on scientific benchmarks. *Software – Practice and Experience*, 33(15):1419–1445, 2003.

[7] C. Ding. Improving software pipelining with unroll-and-jam and memory-reuse analysis. Master's thesis, Michigan Technological University, 1996.

[8] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 68–77, Albuquerque, NM, 1993.

[9] R. Ge. Predicting the effects of register allocation on software pipelined loops. Master's thesis, Michigan Technological University, 2002.

[10] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 258–267, 1993.

[11] M. Lam. Software pipelining: An effective scheduling technique for VLIM machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Designe and Implementation*, pages 318–328, Atlanta, GA, June 1988.

[12] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *the 27th International Symposium on Microarchitecture (MICRO-27)*, pages 63–74, San Jose, CA, 1994.

[13] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 274–286, Paris, France, 1996.