Register Pressure Guided Unroll-and-Jam

Yin Ma Steve Carr

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

Abstract

Unroll-and-jam is an effective loop optimization that not only improves cache locality and instruction level parallelism (ILP) but also benefits other loop optimizations such as scalar replacement. However, unroll-and-jam increases register pressure, potentially resulting in performance degradation when the increase in register pressure causes register spilling. In this paper, we present a low cost method to predict the register pressure of a loop before applying unroll-and-jam on high-level source code with the consideration of the collaborative effects of scalar replacement, general scalar optimizations, software pipelining and register allocation. We also describe a performance model that utilizes prediction results to determine automatically the unroll vector, from a given unroll space, that achieves the best run-time performance.

Our experiments show that the heuristic prediction algorithm predicts the floating point register pressure within 3 registers and the integer register pressure within 4 registers. With this algorithm, for the Polyhedron benchmark, our register pressure guided unroll-and-jam improves the overall performance about 2% over the model in the industry-leading optimizing Open64 backend for both the x86 and x86-64 architectures.

1. Introduction

Unroll-and-jam is a loop transformation that increases the size of an inner loop body by unrolling outer loops multiple times followed by fusing the copies of inner loops back together [4]. Unroll-and-jam is a very effective loop optimization that is used in modern optimizing compilers. A carefully designed unroll-and-jam transformation can dramatically improve single-node loop performance of parallel or sequential code via improved cache and instruction-level parallelism (ILP) [1, 7, 9, 10, 11, 23]. Many other loop optimizations such as loop tiling also integrate unroll-and-jam as a part of their design [19, 23].

The new loop body transformed after unroll-and-jam contains statements from multiple loop iterations. Array references originally located on different loop iterations may share address arithmetic in the unroll-and-jammed loop body. As a result, the live range of an address register often increases after unroll-and-jam. Additionally, references to the same memory location that originally occur on separate outer-loop iterations may now occur in the innermost loop. Scalar replacement will effect register allocation for those array references and increase register pressure.

When a value cannot be allocated to a register it must be spilled back to memory. The performance of a loop may not be improved after optimizations due to increased register pressure from unroll-and-jam. Sometimes performance dramatically degrades because excessive register spilling increases the number of instructions and destroys cache locality. Although microprocessor technology has advanced at an astonishing rate in the past decade, the number of physical registers available for a program can still be considered small. Once register pressure is increased due to loop optimizations, it is difficult to reverse it back later in compilation and code generation. Thus, all optimizations that may increase register pressure should precisely control their register requirement.

To achieve this goal, we present a pseudo-schedule based low cost register prediction algorithm for unroll-and-jam along with a performance model that uses predicted results to determine automatically an unroll vector from a given unroll space that achieves excellent run-time performance. The prediction algorithm has $O(n^2)$ time complexity in practice where n is the size of the loop body.

Given a loop and an unroll vector, the prediction algorithm predicts the register pressure with the consideration of the effects of scalar replacement, general scalar optimizations, software pipelining and register allocation without actually performing any of them. The whole framework is designed for a high-level loop representation such as an abstract syntax tree, providing the compiler a quick and effective way in an early phase to obtain register pressure knowledge available only in the final code generation. Approximation and simplification make the prediction process not only much cheaper than iterative approaches but also allow the process to be used in all kinds of compilation environments. Moreover, since major loop nest optimizations often operate on a high-level representation, our algorithms can also easily be integrated into other loop optimization algorithms that use unroll-and-jam in order to avoid unexpected performance degradation due to high register pressure.

In this paper, we begin with a brief discussion of the previous work on register-pressure prediction. Then, we give a review of unroll-and-jam, scalar replacement and software pipelining. Next, we present the prediction algorithm and the experiment showing the effectiveness of our approach. Finally, we give our conclusion and discuss future work.

2. Previous Work

Wolf et al. [23] present a technique to ensure unroll-and-jam, scalar replacement and software pipelining do not use too many registers. Their method defines the pipeline filling requirements of a processor as the number of floating-point registers required to keep the pipeline running at full speed, which is a reserved value. 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 may overestimate the number of registers required since the registers reserved for pipeline filling may not all be needed.

Carr et al. [7, 12] estimate register pressure before applying scalar replacement with a reserved number of registers for scheduling that is experimentally determined. This value is added into the estimated number of scalars used in a loop by scalar replacement. As in Wolf's method, this technique may reserve more registers than necessary, but also may reserve too few.

Huff [16] defines, two metrics, *MaxLive* and *MinAvg*, to measure the register pressure for software pipelining. MaxLive is only available after creating a software pipeline, making it too late for prediction. MinAvg represents a lower bound on register pressure before pipelining. It 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 and assumes all variables can be scheduled such that the minimum lifetimes are achieved. Under high resource constraints the minimum lifetimes are often not achieved, resulting in highly inaccurate register pressure estimation.

Ding [13] proposes an approach using MinDist to compute register pressure. MinDist is a two dimensional array used to compute dependence constraints in software pipelining 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 is predicted as $\lfloor \frac{lifetime}{II} \rfloor$. MinDist, however, ignores resource constraints, resulting in an imprecise prediction under high resource constraints.

Ge [15] describes a method using 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. However, Ge's method is designed for a low-level language representation instead of the source code level.

Aleta et al [2] use pseudo-schedules to guide data dependence graph partitioning. In this paper, they formally define an approximation similar to Ge's approximate schedules: "The pseudo-scheduler is a simplified version of the full instruction scheduler and estimates key constraints that would be encountered in the final schedule." [2] This property of the pseudo-scheduler makes it a good method to predict the effects of software pipelining. However, their method must compute a pseudo-schedule for every resulting partition. This results in an increase in compile time on the order of a factor of 10.

Carr and Kennedy [11] give an equation-based algorithm to predict register pressure for unroll-and-jammed loops. This algorithm computes register pressure from updated dependences on array references. It can predict the number of floating-point registers and address registers used given an unroll vector. This method computes the register pressure of arithmetic computations using Sethi-Ullman numbering. Sethi-Ullman numbering only gives the minimum number of registers possible and ignores scheduling effects, resulting in a very low prediction accuracy for integer register pressure.

Triantafyllis et al. [22] present the first iterative compilation technique suitable for general-purpose production compilers, called optimization-space exploration (OSE). OSE achieves much faster speed over the traditional iteration approaches by using pre-optimization predictions to reduce the exploration space and using a static performance estimator to avoid running the program for measuring actual run-times. Since OSE is a heuristic method, it may not reach the best configuration due to the tradeoff in compile-time savings.

Ma et al. [20] present a pseudo-schedule based low cost algorithm to predict the register pressure of a loop before applying scalar replacement on high-level source code. In the algorithm, a fast constructor scans source code and creates a data dependence graph (DDG) on the fly to reflect all potential future changes in later compilation phrases. A pseudo-scheduler predicts register pressure from this DDG. It can predict floating-point and integer register pressure within less than 3 registers with a time complexity of $O(n^2)$ in practice and a worst case complexity of $O(n^3)$. This method exposes a potential framework to predict register pressure for other source code level loop optimizations. However, predicting the changes made by scalar replacement only involves removing or changing array references in a loop body. No work has been done to account for structural changes in a loop due to unroll-and-jam. In addition, there has been little work in register-pressure guided high-level loop optimization. This paper addresses both of these issues.

3. Background

In this section, we review unroll-and-jam, scalar replacement and software pipelining. We discuss how these transformations modify a loop body in detail as well as their effects on register pressure.

3.1 Unroll-and-Jam

Unroll-and-jam is the combined operations of loop unrolling and jamming [4]. For example, the following loop contains two array references:

```
DO I = 1, N*2
  DO J = 1, M
    A(J,I) = A(J-1,I) + B(J)
  ENDDO
ENDDO
```

If we unroll the `I`-loop by a factor of two, after the two new `J`-loops are jammed together, the loop becomes:

```
DO I = 1, N*2, 2
  DO J = 1, M
    A(J,I)   = A(J-1,I) + B(J)
    A(J,I+1) = A(J-1,I+1) + B(J)
  ENDDO
ENDDO
```

Jamming must maintain the order of data dependences. If it reverses the execution order of array references, jamming becomes illegal[7]. Unrolling the innermost loop is always legal because no jamming is required. Usually, unroll-and-jam only refers to the unrolling of loops nested outside of the innermost loop. However, our prediction algorithm assumes the innermost loop can also be unrolled, providing a unified solution for loop unrolling.

The number of times a loop is unrolled is called the *unroll factor*. To represent multiple unroll factors, we use an *unroll vector* with one unroll value for each nesting level of the loop.

After unroll-and-jam, more array references are available in the innermost loop, providing scalar replacement more opportunities to convert memory references into scalars and software pipelining

more instruction-level parallelism. By itself, the reuse of array addressing computations also reduces execution time. However, the increased register exploitation demands more physical registers, often leading to spilling in register allocation. Too much spilling results in performance degradation due to increased memory accesses and increased cache interference.

3.2 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 [6, 8, 7, 12, 14]. Combined with unroll-and-jam, scalar replacement can improve the efficiency of pipelined functional units more than by itself. Considering the unrolled loop in the previous section, there are six memory references and two floating-point additions in the innermost iteration. The value referenced by `A(J-1,I)` and `A(J-1,I+1)` are defined one iteration of the `J`-loop earlier by `A(J,I)` and `A(J,I+1)`, respectively. Using scalar replacement to expose the reuse, the resulting code is:

```
DO I = 1, N*2, 2
  a0 = A(0,I)
  a1 = A(0,I+1)
  DO J = 1, M
    b0 = B(J)
    a0 = a0 + b0
    A(J,I) = a0
    a1 = a1 + b0
    A(J,I+1) = a1
  ENDDO
ENDDO
```

Here the number of memory references decreases to three with the number of floating-point arithmetic operations remaining the same, which removes one more memory reference than when only applying scalar replacement. If the original loop is bound by memory accesses, unroll-and-jam and scalar replacement improve performance. However, three additional scalars are used, demanding more registers to hold values. Thus scalar replacement increases register pressure and may cause excessive register spilling and degrade performance.

3.3 Software Pipelining

Software pipelining [3, 18, 21] is an advanced scheduling technique for modern processors. Modulo scheduling [18, 21] is one popular approach to software pipelining. This approach tries to use the minimum possible cycles to schedule one iteration of a loop such that no resource and data 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 represents the number of cycles required to execute a single iteration of a loop. Given a loop and a target architecture, the resource initiation interval (ResII) gives the minimum number of cycles needed to execute one iteration of 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 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 II.

Software pipelining can significantly improve loop performance. Consider a loop L that iterates n times and contains three instructions: A, B, and C. When we assume the dependences in L require a sequential ordering of these operations within a single loop iteration, even if the target architecture allows 3 operations to be issued in a CPU cycle, the schedule for L still requires 3 cycles to finish one iteration due to those dependences. The loop L would execute in $3 \times n$ cycles on a machine with one-cycle operations.

A software pipelined version of the loop L might well issue all three operations in one cycle by overlapping executions from different loop iterations. Under ideal circumstances, the scheduled loop consists of a single-instruction loop body $A_{i+2}B_{i+1}C_i$ where X_j denotes operation X from iteration j of the loop [3]. The cost of the software pipelined version is about one-third of the cost of the original one, namely $n + 2$ cycles including the prelude and postlude to fill and drain the pipe. So software pipelining can, by exploiting inter-iteration concurrency, dramatically reduce the execution time required for a loop.

Unfortunately, the overlapping of loop iterations leads to additional register requirements. For illustrative purposes, we assume that operation A computes a value, v, in a register and that operation C uses v. In the initial sequential version of the loop L, 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 multiple loop iterations are executed simultaneously. In this particular case, we would need two more registers for storing the extra two copies of v.

Unroll-and-jam, scalar replacement and software pipelining are all optimizations that may increase register pressure. If their collective effects are not considered, the predicted register pressure cannot be accurate. In our prediction algorithm, the combined effects are handled in one step. In the next section, we detail the entire prediction algorithm.

4. Register Pressure Prediction Algorithm

Source code is lowered to a low-level intermediate form before register allocation. In order to predict register pressure precisely, it is critical to have prediction algorithms work on a representation close to what the register allocator sees. A data dependence graph (DDG) not only contains all expressions but also the relationships among data, providing all information needed to pass into a scheduler, then later an allocator. So the first step of our prediction algorithm is to construct a DDG that reflects all changes made by as many optimizations as possible.

4.1 DDG Construction

Our prediction algorithm is applicable to innermost loops that contain assignment statements with array references, scalars and arithmetic operators. For one loop nest, our approach models unroll-and-jam applied to at most two outer loops. While this restriction is not necessary, it simplifies the approach since it is rare to apply unroll-and-jam to more than two loops. Our prediction algorithm considers the effects of unroll-and-jam, scalar replacement, software pipelining and general scalar optimizations.

Constructing the DDG for a loop nest consists of four major steps applied in order:

1. Create the base DDG for the loop before unroll-and-jam.
2. Prepare the DDG after unroll-and-jam by duplicating the base DDG.
3. Delete unnecessary DDG nodes and edges.
4. Generate cross-iteration edges.

In the base DDG constructed from the source code, nodes represent each elementary operation constructed and edges represent data dependences[17]. Each edge is labeled with the vector $\langle$*Delay*,*Diff*$\rangle$, where *Delay* represents the minimum number of cycles needed to perform the source operation and *Diff* is a vector that represents the number of loop iterations for each containing loop between the source and the sink of the edge.

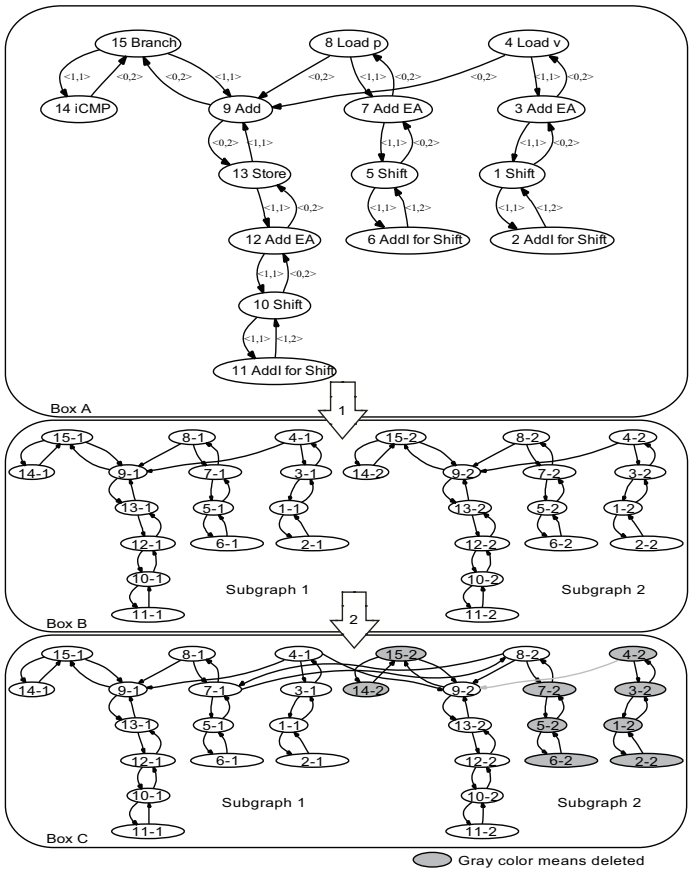

Figure 1. The DDG Construction Process of the Example Loop

4.1.1 Preparing the DDG

Unroll-and-jam brings outer iterations closer together within the innermost loop. Each copy of the innermost loop body basically contains the same expressions except for the changes in the index expressions of arrays. If a DDG is built from an unroll-and-jammed loop body without considering scalar replacement, the resulting graph consists of many similar subgraphs. Each subgraph corresponds to one copy of an unrolled iteration. The edges that cross subgraphs are the data dependences between operations in different copies of the innermost loop bodies. Although this DDG contains many redundancies not found in an optimized DDG, in order to simplify the DDG construction process, we first generate the DDG from the loop before unroll-and-jam as the base DDG then duplicate this DDG multiple times based on the given unroll vector. Later, we delete nodes to reflect the changes by scalar optimizations. At the end, we compute cross-subgraph edges and add them to get the final DDG.

Ma et al. [20] present a low-cost algorithm to construct the base DDG. The time complexity of the duplication step is $O(n)$ where n is the size of the input loop. For example, to construct the DDG for the following loop:

```
DO J = 1, N
  DO I = 1, N
    U(I,J) = V(I) + P(J,I)
  ENDDO
ENDDO
```

We first create the base DDG shown in the Box A of Figure 1 using the low-cost algorithm developed by Ma et al. [20]. Note that the labels on the edges represent the value of *Diff*. If the `J` loop is unrolled by a factor of 2, the loop becomes

```
DO J = 1, N, 2
  DO I = 1, N
    U(I,J) = V(I) + P(J,I)
    U(I,J+1) = V(I) + P(J+1,I)
  ENDDO
ENDDO
```

We must duplicate the base DDG once for the copy of the innermost loop body. Box B in Figure 1 contains the updated DDG, containing subgraph 1 and subgraph 2. Clearly, this DDG is not precise since it contains redundancies that will be eliminated by other optimizations. In the next section, we discuss a marking algorithm that will update the DDG to reflect the effects of scalar optimizations.

4.1.2 Deleting Nodes and Edges

During the construction of the base DDG, the effects from scalar replacement, general scalar optimizations and software pipelining have been considered. But after duplication, we must reconsider them because unroll-and-jam usually provides more opportunities for those optimizations. In our prediction algorithm, the nodes in a DDG are classified into four categories: load and store nodes, array addressing nodes, arithmetic nodes and loop control nodes, which are each handled separately. The algorithm for node deletion is given in Figure 2. The input `V` is the unroll vector corresponding

to the subgraph G. For the example loop, the function *deleteNode* will be called twice, where V is $\langle 1, 1\rangle$ and $\langle 1, 2\rangle$.

```
deleteNode( Subgraph G, UnrollVector V ){
  foreach Load/Store node n ∈ G
    foreach true/input dependence, e, incident on n
      d = the distance vector of e
      D = the direction vector of e
      if d is available then
        update d using V
        if d becomes loop independent then
          recursively delete n and dead addressing nodes for n
      else if D can be handled then
        check D
        if D is deletable then
          recursively delete n and dead addressing nodes for n
    if n is still not dead then
      r = analyzeSubscript
      if r is REMOVEALL then
        delete addressing nodes for n
      else if r is PARTIALREMOVE then
        delete part of addressing nodes for n
  if G is not the last subgraph then
    delete all loop control nodes
}
```

Figure 2. DDG Node Deletion Algorithm

Loop control nodes contain branch nodes and their supporting comparison nodes. For the DDG in the Box A of Figure 1, nodes 14 and 15 are loop control nodes. Since only one set is required in all iteration copies, the algorithm just keeps the set in the last subgraph and marks the others as deleted. Therefore, nodes 14-2 and 15-2 are marked as deleted in the Box C. The nodes shaded gray in Figure 1 have been deleted. In our algorithm, all arithmetic nodes are always kept because they hold the logic of a program and it is not common for subexpressions to exist amongst array operations. In most cases, they should not be changed during any compilation. In the Box A, node 9 is an arithmetic node.

Load and store nodes usually correspond to array references. In the Box A, nodes 4, 8 and 13 are in this category. Nodes 1, 2 and 3 are the addressing nodes for the array `V(I)`; nodes 5, 6 and 7 are the addressing nodes for the array `P(J,I)` and nodes 10, 11 and 12 are for the array `U(I,J)`. Array references with incoming loop independent true or input dependences are removed by scalar replacement. Nodes for this kind of array are not generated during the base DDG construction. Scalar replacement also applies to any node having an innermost loop carried incoming dependence. These kinds of nodes exist in the base DDG. Unrolling the innermost loop can remove the innermost loop carried dependence if the corresponding distance is less than the unroll amount. To simplify the computation, we only delete the nodes for array references having loop independent incoming dependences created by unroll-and-jam that cross copies of the innermost loop in the algorithm. As a tradeoff, the final DDG may still contain some nodes carrying the innermost loop dependence that should be removed by scalar replacement, resulting in lower precision. If the distance vector d of a dependence is available, Carr and Kennedy give an equation to predict the updated dependence vector with exact values after unroll-and-jam [11]. With this equation, if any array has an incoming dependence that is loop independent in any iteration copy, indicated by all items of d becoming 0, its corresponding load or store node should be deleted.

When only the direction vector D is available for an array reference, if D only contains $=$ and $*$, where $*$ is caused by this array reference being invariant with respect to the corresponding loop, deleting this array can be determined using the rules:

- The node with D as its incoming direction vector is deletable if the D_i is $*$ and the unroll factor for loop i is greater than one.
- Otherwise, the node is not deletable

All unqualified array references will be retained. In the example loop before unroll-and-jam, `V(I)` has a direction vector of $\langle *, =\rangle$ where the `J`-loop entry is $*$. When the `J`-loop is unrolled by a factor of two, the second copy of `V(I)` will be deleted because the unroll factor of the `J`-Loop for this copy is 2. This copy will be removed by scalar replacement. Thus, node 4-2 is marked as deleted.

After deleting all load and store nodes that should be deleted, their corresponding addressing computation nodes may be dead. Those dead nodes can be deleted recursively by checking nodes starting from a deleted load/store node via incoming edges. If all outgoing edges of a node are deleted, this node is marked as deleted. In the example after the load node 4-2 is deleted, its supporting addressing node, node 3-2 is not used. Therefore, node 3-2 is marked as deleted. Recursively, nodes 1-2 and 2-2 are also marked as deleted.

Besides the deletion caused by data dependences, reused addressing computation also results in the removal of some nodes corresponding to addressing computation. If an array has a format like $A(h, \ldots, i, \ldots, j, \ldots, k)$ in Fortran and all its dimensions are constant in the current code region, we use the function *analyzeSubscript*(A, L_1, L_2, j) below to determine how to process is addressing computations, where A is an array, L_1 and L_2 are the unrolled loops, and j is the induction variable of the innermost loop.

```
analyzeSubscript( A, L1, L2, j ) {
  p = the induction variable of L1
  q = the induction variable of L2
  if p is on the right of q in A's subscript then
    swap p, q and L1, L2
  if p has unroll factor larger than 1 then
    if p is the leftmost and on the left of j then
      return REMOVEALL
    else if p is the leftmost and p is j then
      return REMOVEALL
    else if p is on the left of j but is not j then
      return PARTIALREMOVE(p TO j )
    else return KEEPALL
  return KEEPALL
}
```

Figure 3. Algorithm for Analyzing Array Subscripts

In the algorithm *analyzeSubscript*, p always represents the leftmost unrolled induction variable in the subscripts of A. [1] Because any node invariant with respect to the innermost loop is not generated in the base DDG, the addressing computation nodes from subscript h to j appear in the base DDG if j is the innermost loop induction variable. For example, considering `U(I,J)` in the example loop, nodes 10, 11, and 12 are the addressing computation only from subscript `I`. When an outer loop p is unrolled, if p is at a location like k, all addressing nodes for this array in all iterations are retained. So, if we unroll loop `J` for `U(I,J)`, nodes 10, 11, and 12 should all be retained in the subgraphs. If p is at j, which means p is the innermost induction variable, all addressing nodes are retained. If p is at h, all addressing nodes except for the copy for the first iteration should be deleted since p in the leftmost subscript and the addressing computation can share the same portion from h to j. `P(J,I)` represents this case in Box C of Figure 1. If p is at i, because p is not in the leftmost subscript, the addressing

[1] This discussion assumes column-major ordering. Simple modifications are necessary for row-major ordering.

nodes corresponding the indices from i to j should be deleted except for the one in the first iteration. However, the addressing nodes corresponding to the indices from h to i should be retained for all copies. To support this case, each subscript should be tagged with its corresponding addressing nodes during construction of the base DDG. If a compiler optimizes subscripts using a different approach than outlined here, in order to make prediction accurate the DDG modification should be adjusted to reflect the differences.

If two loops are unrolled, even if loop q has an unroll factor larger than 1, p still determines the final results in the DDG. Considering the example loop when both loop `I` and `J` are unrolled by a factor of two, there are four updated copies of `P(J,I)`:`P(J,I)`, `P(J+1,I)`, `P(J,I+1)`, `P(J+1,I+1)`. The corresponding values of the unroll amounts for p and q are $\langle 1,1\rangle, \langle 2,1\rangle, \langle 1,2\rangle$, and $\langle 2,2\rangle$. The addressing computation nodes in the two subgraphs where `P(J,I)` and `P(J,I+1)` are located will be retained, but the rest will be deleted because p has an unroll factor no larger than 1 only in these two cases. If a subscript contains another array reference, it should be treated as an unrolled variable that changes as long as it contains any induction variable that is currently being unrolled. Otherwise, it should be treated as a constant even if it contains the innermost induction variable.

4.1.3 Create Cross-Iteration Edges

In order to get a precise DDG, nodes are marked to reflect the changes from scalar replacement, value reuse and other optimizations. The duplication of the base DDG creates the edges inside one iteration but no cross-iteration edges. The scheduler applied later assigns the location of a node based on the *Delay* and *Diff* tags on its edges. Without cross-iteration edges, all subgraphs are always scheduled from cycle zero, causing an overestimation of register pressure. Based on the equation from Carr and Kennedy [11], cross-iteration dependence edges can be computed by scanning each dependence edge in the base DDG. This process can also be integrated into the node deletion step. If the source and sink node of an updated dependence are both live, we add a corresponding edge into the DDG. If the sink node is deleted, all nodes to which it points become the new sink nodes. In the example loop, when node 4-2 is deleted, we know the dependence causing the deletion is from the node 4-1. At this time, node 4-1 is live and node 9-2 with which node 4-2 is originally connected is also live. So an edge from 4-1 to 9-2 is added in the Box C of Figure 1. Node removal caused by the reuse of address computations also creates cross-iteration edges. The node that receives a value from a node deleted in its iteration copy should link to the same node that generates the same value located in the previous iteration copy. The new edges between 7-1 and 8-2 belong to this case in the Box B of Figure 1.

4.2 Register Prediction

One cornerstone of a successful prediction is a DDG that can precisely reflect the code right before register allocation. The previous section has given an algorithm to achieve this goal. The next phase is to predict register pressure. Based on previous research, predicting register pressure using a pseudo-schedule is a promising method because it considers the impact from instruction scheduling. Thus, our algorithm also creates a pseudo-schedule to predict register pressure.

The pseudo-scheduler is a fast scheduling algorithm that does not consider back edges in the DDG. It schedules using a depth-first scan of a graph starting from the first node of the first iteration copy. Initially, the algorithm tries to schedule the loop using a schedule length of MinII. Although it has been proved that unroll-and-jam does not change the RecII of a loop, the interactions between scalar replacement and address arithmetic optimization may yield a different RecII in the low-level code when compared with the RecII predicted from the constructed DDG. However, a large unroll amount causes the ResII of the loop to become the dominant factor over RecII. Moreover, computing RecII is very costly, at least $O(n^3)$ for a precise value or even a worst case $O(n^2)$ for an approximate value using the algorithm presented in [20]. Thus, we only compute RecII for the base DDG using the approximation algorithm and use this value as the RecII in the computation of the estimated II.

ResII is computed from the final DDG. The maximum value of RecII and ResII becomes the estimated II. After all nodes are scheduled, we use the same DU chain estimation method used in [20] to get the final predicted register pressure.

4.3 Further Simplification

If nodes are scheduled in the order of iteration copies, it is possible to simplify the DDG construction algorithm and pseudo-scheduler further with a lower prediction cost without losing too much precision. The biggest benefit of this simplification is to make the DDG construction and pseudo-scheduler operate incrementally so that the whole framework can run efficiently with a large unroll space or a large loop body. In a scheduler, if a scheduled loop is viewed as a set of nodes from different iteration copies that are distributed into the cycle slots of a schedule table, the majority of the nodes from an iteration copy usually are clustered together. Considering the cycle timeline, the nodes from an iteration copy A are usually scheduled before the nodes from another iteration copy B if A is before B in the loop body. Based on these two observations, it is reasonable to ignore the generation of cross-iteration edges and only mark nodes in the DDG construction. During pseudo-scheduling, the subgraphs of iteration copies are scheduled one by one in order. For each subgraph, we apply the same depth-first scheduling algorithm. For most cases, a pseudo-scheduled loop with the simplified algorithms yields good results.

With these two simplifications, the DDG constructed and the pseudo-schedule created by a unroll vector with a smaller unroll amount can be utilized again with a unroll vector with a larger unroll amount, reducing the whole prediction cost significantly in register pressure guided unroll-and-jam since the nodes and schedule of one iteration copy are same for predicting any unroll vector.

5. Register Pressure Guided Unroll-and-jam

Our register pressure prediction algorithm can be directly integrated into any loop optimization strategy utilizing unroll-and-jam. The estimated register pressure can be used in the constraint functions for those loop optimizations. Sometimes, pursuing the minimum register pressure can maximize the final performance, especially when not many registers are available in a processor. So we present a performance model to guide unroll-and-jam using our prediction algorithm.

To measure the performance of an unroll-and-jammed loop, we use the *unitII* as the indicator, where the unitII is defined as $\frac{II}{u}$ where II is the II of the innermost loop and u is the total unroll amount, *i.e.*, the product of the unroll factors. This leaves the objective of our performance model as minimizing the unitII with the smallest unroll amount. Given a loop nest, the model assumes up to two loop levels can be picked to do unroll-and-jam since applying unroll-and-jam to more than two loops is rarely possible and profitable.

While scanning a pre-defined unroll space, the estimated unitII is computed for each unroll vector. Usually, unroll vectors are processed from small to large consecutively in order to utilize the incremental property of the simplified prediction algorithm. The unroll vector with the minimum unitII is selected for the final

application of unroll-and-jam. If there is more than one unroll vector with the same unitII, the one with the smallest total unroll amount is selected.

5.1 Computing UnitII

After the loop levels are chosen, the next step is to estimate the unitII. For a certain unroll vector, the II computed in the pseudo-scheduler is the estimated II without any spilling. Spilling instructions consume CPU cycles and perhaps delay the execution of other memory instructions due to cache misses. At the source code level, it is difficult to determine where or how many spilling instructions will be inserted in the assembly code. However, if the number of nodes in two graphs is identical, the graph that contains more nodes with a higher degree of outgoing edges has a higher possibility of resulting in more spilling instructions. This is because when the result of a node is spilled into memory, spilling instructions for loading the value back are required for every node connected by its outgoing edges. So, we use the following equations to estimate a unitII for particular unroll vector:

$$unitII = \frac{II + IIPenalty_i + IIPenalty_f}{TotalUnrollAmount}$$
$$IIPenalty_i = \frac{(R_i - P_i) \times (D_i + E_i) \times A}{N_i}$$
$$IIPenalty_f = \frac{(R_f - P_f) \times (D_f + E_f) \times A}{N_f}$$

Where R is the number of registers predicted and P is the number of registers available. D is the total outgoing degree and E is the total number of cross iteration edges. A is the average memory access penalty, representing the estimated cycles of a single spilling instruction under the influences from instruction execution, cache performance or pipeline delay, etc. N is the number of nodes in a DDG. The subscript i indicates a variable is an integer and f indicates the variable is floating point. The unroll vector with the smallest unitII and the smallest total unroll amount is used to perform unroll-and-jam.

6. Experiment

Our experiment contains two parts. One is to test the accuracy of our register pressure prediction algorithm. Another is to measure the performance of register guided unroll-and-jam. The register pressure prediction algorithm has been implemented in a source-to-source Fortran compiler. The performance of the register pressure algorithm described in this section is also measured based on the simplified version without considering cross iteration edges. The compiler optimizes loop nests by unroll-and-jam and scalar replacement, using a number of other auxiliary transformations needed for dependence analysis [5]. The code processed by the source-to-source compiler is fed into a retargetable compiler for ILP architectures. The retargetable compiler performs general machine-independent scalar optimizations including constant propagation, global value numbering, partial redundancy elimination, strength reduction and dead code elimination. The software pipelining algorithm used is iterative modulo scheduling [21].

The target architecture has two integer functional units and two floating point functional units, sharing 16 floating-point registers and 16 integer registers. All instructions have a latency of two cycles. An infinite number of available physical registers is assumed in the backend in order to count register pressure. The register pressure of a loop is the sum of registers created in the innermost loop and its live-in registers. In this experiment, we extract 13 loop nests from the SPEC2000 benchmark suite as test loops. Loops 01 and 02 are extracted from 171.swim and loops 03, 04 and 05 are extracted from 172.mgrid. The rest of loops are from 200.sixtrack. All loop nests contain at least two levels and various combinations of dependences and arrays in order to cover more situations. All are also amenable to unroll-and-jam, scalar replacement and iterative modulo scheduling.

6.1 Accuracy of Register Pressure Prediction

For a loop, if we predict register pressure without any unroll-and-jam, our algorithm will generate results identical to the PRP algorithm presented in [20]. PRP gives better register-pressure prediction precision on high-level source code than Ge's method, Huff's MinAvg approach and Ding's MinDist technique. Table 1 shows the performance of PRP on our test suite without unroll-and-jam.

	Integer	Floating Point
Average Err	1.73	0.55
Relative Err	0.13	0.22

Table 1. The Performance of PRP Without Unroll-and-Jam

In Table 1, "Average Err" means the average of the absolute values of the difference between the predicted register pressure and the actual register pressure of the final code. "Relative Err" is the average of the absolute values of the difference between the predicted register pressure and the actual register pressure divided by the actual register pressure. Our algorithm predicts the integer register pressure within 1.73 registers on average and the floating-point register pressure within 0.55 registers.

To test the accuracy of our prediction algorithm for unroll-and-jam, we compare our predicted register pressure with the equation-based unroll-and-jam register prediction algorithm presented in [11]. In the experiments, no registers are reserved in order to do a comparison of the accuracy of the prediction methods. The outermost two loop levels are unrolled. The unroll space contains 19 combinations of unroll amounts such that the total unroll amount is no larger than 12 and the single unroll amount for one loop level is no more than 5. Thus, the unroll vectors are from $\langle 1, 1\rangle$ to $\langle 1, 5\rangle$, from $\langle 2, 1\rangle$ to $\langle 2, 5\rangle$, from $\langle 3, 1\rangle$ to $\langle 3, 4\rangle$, from $\langle 4, 1\rangle$ to $\langle 4, 3\rangle$ and from $\langle 5, 1\rangle$ to $\langle 5, 2\rangle$, where the left value is for the outermost loop level. For the the 247 unroll vectors applied to the 13 loop nests in our suite, the average off achieved by the prediction algorithm is shown in Table 2.

Our algorithm gets an overall average off of 3.97 for integer register pressure prediction and 2.92 for floating-point prediction. Both values are better than Carr and Kennedy's algorithm, especially for the integer portion. All prediction results with unroll-and-jam are worse than those without unroll-and-jam. The larger the unroll amount, the lower the accuracy. This fact is explained by the approximations contained in the algorithm. The ignoring of cross iteration edges in constructing the DDG and backward edges in pseudo-scheduling causes the predicted values to be off from the real values. Since all iteration copies derive from one base copy, the error that exists in the base copy is accumulated while unrolling. Cross iteration edges and backward edges are used to control the scheduled location of a node. Without them, a node in the pseudo-schedule may be scheduled earlier than what it should be in the real schedule. As more iteration copies are scheduled, more cross iteration and backward edges are ignored, causing lower prediction accuracy.

In this experiment, register pressure increases with any unroll amount bigger than one for all loop nests. If we consider the sequence of register pressure values for each successive unroll vector, usually a repeated pattern is observed in the register pressure change between two consecutive unroll vectors. For example, loop 08 requires 10, 16, 21, 24, and 26 registers corresponding to unroll vectors $\langle 1, 1\rangle$, $\langle 1, 2\rangle$,$\langle 1, 3\rangle$,$\langle 1, 4\rangle$ and $\langle 1, 5\rangle$. The incremental values for each step are 4, 5, 3, and 2. The average trend for the increase of one step is approximately 4. The formation of the repeated pattern is from the similarity of iteration copies. At first we assume

Unroll Amount	Our Algorithm				Carr and Kennedy Algorithm			
	Integer		Floating Point		Integer		Floating Point	
	Avg Err	Rel Err	Avg Err	Rel Err	Avg Err	Rel Err	Avg Err	Rel Err
2	2.48	0.15	1.36	0.34	9.24	0.74	2.27	1.16
3	2.80	0.15	2.24	0.38	11.24	0.68	1.08	0.43
4	3.28	0.16	3.03	0.49	11.84	0.64	2.44	0.56
5- 6	3.80	0.18	3.40	0.48	13.08	0.65	3.00	0.63
7 - 9	4.58	0.20	3.67	0.49	14.63	0.67	3.58	0.66
10 - 12	5.67	0.24	3.79	0.47	16.06	0.65	4.42	0.95
1 - 12	3.97	0.18	2.92	0.44	14.2	0.66	3.54	0.7

Table 2. The Detailed Performance Comparison Table on Register Prediction

there is no node reuse or replacement. At each point where an unroll factor is increased by one, unroll-and-jam will create an iteration copy with the same data dependence subgraph. During scheduling, almost the same number of extra registers is required for scheduling the subgraph, especially, when ResII dominates. Usually, the nodes affected by scalar replacement only occupy a small portion of the total nodes. Those replacements only happen in regular and repeated locations based on the property of updated dependences. For the address arithmetic reuse, the same situation exists, resulting in a repeated pattern. As a result, for a set of consecutive unroll vectors, the incremental or decremental change of register pressure is approximately a constant value. We call this kind of phenomena for a loop nest a trend. The quantity of a trend is the value of that constant. We found that if our prediction algorithm accurately captures the trend, the algorithm performs well.

6.2 Performance of Register Pressure Guided Unroll-and-Jam

In order to know the real performance of our new model, we implemented our model along with all register pressure prediction algorithms in the Open64 backend. Open64 is an open-source optimizing compiler. The commercial FORTRAN compilers that utilize the Open64 backend have been proved to deliver the fastest code in the world on the Polyhedron benchmark from Polyhedron Software Ltd. This benchmark suite is designed in FORTRAN and used for comparing the performance of the latest commercial FORTRAN compilers on x86-64 and x86 architectures. It only allows one option against all benchmarking programs so it is an ideal suite to evaluate if our loop model can automatically pick up the best unroll number and deliver a good performance.

Our implementation replaces the register pressure prediction algorithm and unroll-and-jam loop model used in the Open64 backend directly by our register pressure prediction algorithm and our loop model except for one change: the $UnitII$ variable computed in our loop model will add an extra value C computed with the information from the Open64's cache model. Since the units are different between our loop model and Open64's cache model and our loop model should perform the dominate effect, the equation to compute C is defined as $C = \lceil \frac{CacheCost}{UnitII} \rceil^{\frac{1}{10}}$, where $CacheCost$ is the $cycles\_per\_iter$ variable defined by Open64's cache model. Compared with $UnitII$, C is always a relative small number. $CacheCost$ contains the hints of loop header cost and cache penalty cost. So it is good to add C with $UnitII$ and the original $UnitII$ still dominates the computation.

In all experiments, we used the Polyhedron standard benchmark mode to evaluate performance for each loop model. The machine running the benchmark suite has dual AMD Opteron 2216 processors with 4GB of memory. The operation system is the 64-bit Fedora 7. The highest optimization level is always used in this experiment along with the same machine description. The detailed result data are shown in Tables 3 and Table 4.

In the tables, 'Compile' means the time used to compile a test and 'Ave Run' means the execution time reported by the Polyhedron benchmark. Our model achieves over two percent average speedup over Open64's model on the geometric mean of the average execution time in both the 32-bit and the 64-bit mode. Performance is improved for most tests. For four tests, the speedup is more than 6%. Only seven tests have worse performance because of the inaccurate prediction results. The compilation speed is increased by an average 2.3%. It is reasonable because our prediction algorithm is more sophisticated than the one used in the Open64 backend. We also believe this is an acceptable tradeoff with the improved performance.

Most of loop optimizations are performed interactively in Open64's compiler. When the highest optimization level is used, some loop optimizations are also applied in the code generation phrase. The unroll-and-jam model is designed at the position before many other high level loop optimizations. If a loop is unrolled to a certain shape, the following loop optimizations may not be triggered or select different algroithms for it. Thus, although our unroll-and-jam model executes a longer time than Open's model, the overall execution time used for the whole compilation may be shortened in some situations. This explains why AIR used less compilation time with our new model in 64-bit mode.

In Open64's loop optimization framework, the unroll-and-jam model and the cache model along with loop permutation are two separate phases. As mentioned, we add an extra value produced by Open64's cache model to the UnitII to reflect the effects of cache. We believe that we could achieve better results with a tighter integration between our unroll-and-jam and Open64's cache model.

7. Conclusion

We have presented an algorithm for register-pressure guided unroll-and-jam. The framework targets prediction from a high-level representation such as an abstract syntax tree. Without applying any actual optimization, the prediction algorithm predicts register pressure after unroll-and-jam, scalar replacement, general scalar optimizations and software pipelining. It constructs a data dependence graph close to the intermediate code right before register allocation and then applies a pseudo-schedule to predict register pressure. The performance model utilizes the predicted register pressure information with a given a unroll space heuristically to determine the unroll vector that may give the best unitII.

In our experiments, the prediction algorithm achieves an overall average off 3.97 for integer register pressure and 2.92 for floating-point registers. For the Polyhedron benchmark, our register pressure guided unroll-and-jam improves the overall performance by about 2% over the model used by the industry-leading optimizing Open64 backend for both the x86 and x86-64 architectures.

Along with good performance, the low cost design and optional simplification make the approach portable and easily integrated into other loop optimizations using unroll-and-jam. We believe a

32bit	Open64's Model		Our Model		Speedup	
Unit: Seconds	Compile	Ave Run	Compile	Ave Run	Compile	Ave Run
AC	2.21	13.45	1.98	13.20	10.41%	1.86%
AIR	7.31	20.79	7.20	20.55	1.50%	1.15%
AERMOD	140.89	35.83	145.57	34.48	-3.32%	3.77%
DODUC	20.78	50.14	21.37	50.25	-2.84%	-0.22%
LINPK	1.02	33.05	1.17	32.97	-14.71%	0.24%
MDBX	3.96	21.68	3.95	21.21	0.25%	2.17%
TFFT	0.83	9.70	0.99	9.26	-19.28%	4.54%
CAPACITA	4.26	75.08	4.86	74.67	-14.08%	0.55%
CHANNEL	2.09	20.35	1.96	19.47	6.22%	4.32%
FATIGUE	6.46	8.40	6.91	7.96	-6.97%	5.24%
GAS_DYN	4.53	9.62	4.76	9.46	-5.08%	1.66%
INDUCT	11.97	34.20	11.94	34.87	0.25%	-1.96%
NF	1.74	31.10	1.52	30.68	12.64%	1.35%
PROTEIN	6.23	59.37	7.57	59.50	-21.51%	-0.22%
RNFLOW	8.84	36.45	9.88	35.84	-11.76%	1.67%
TEST_FPU	5.73	24.95	4.81	23.28	16.06%	6.69%
Geometric Mean	5.18	25.10	5.32	24.57	-2.69%	2.08%

Table 3. Polyhedron 32bit Benchmarking Results

64bit	Open64's Model		Our Model		Speedup	
Unit: Seconds	Compile	Ave Run	Compile	Ave Run	Compile	Ave Run
AC	1.89	11.20	1.88	10.74	0.53%	4.11%
AIR	22.47	18.05	13.89	16.36	38.18%	9.36%
AERMOD	304.31	34.43	325.49	35.46	-6.96%	-2.99%
DODUC	27.91	44.52	28.11	44.76	-0.72%	-0.54%
LINPK	1.21	33.33	1.15	33.16	4.96%	0.51%
MDBX	3.57	20.73	3.99	20.21	-11.76%	2.51%
TFFT	1.14	9.53	1.31	9.18	-14.91%	3.67%
CAPACITA	5.43	69.90	6.12	70.08	-12.71%	-0.26%
CHANNEL	1.68	18.98	1.99	18.74	-18.45%	1.26%
FATIGUE	8.17	6.71	8.62	6.27	-5.51%	6.56%
GAS_DYN	5.78	6.66	5.99	6.56	-3.63%	1.50%
INDUCT	17.85	32.92	17.56	33.00	1.62%	-0.24%
NF	1.32	30.58	1.61	30.55	-21.97%	0.10%
PROTEIN	9.17	55.04	10.00	54.54	-9.05%	0.91%
RNFLOW	8.76	33.07	8.48	31.00	3.20%	6.26%
TEST_FPU	6.21	24.08	5.79	24.03	6.76%	0.21%
Geometric Mean	6.45	22.80	6.59	22.32	-2.10%	2.11%

Table 4. Polyhedron 64bit Benchmarking Results

loop optimization insensitive to register pressure is incomplete. In our research, we have presented prediction algorithms for software pipelining, scalar replacement and unroll-and-jam. In the future, more loop optimizations will be covered.

References

[1] A. Aiken and A. Nicolau. *Loop quantization: An analysis and algorithm*. Tech. Rep, Cornell Univ., Ithaca, N.Y.

[2] A. Aleta, J.M. Codina, J. Sanchez, A. Gonzalez, and D. Kaeli. Exploiting pseudo-schedules to guide data dependence graph partitioning. In *2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 274–286, Charlottesville, Virginia, 2002.

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

[4] F. Allen and J. Cocke. *A catalogue of optimizing transformations*. In Design and Optimization of Compilers, Prentice-Hall, Englewood Cliffs, N.J.

[5] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publisher, 2001.

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

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

[8] 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.

[9] S. Carr. Combining optimization for cache and instruction-level parallelism. In *Proceedings of the 1996 Conference on Parallel Architectures and Compiler Techniques*, pages 238–247, Boston, MA, October 1996.

[10] S. Carr and Y. Guan. Unroll-and-jam using uniformly generated sets. In *Proceedings of the 30^{th} International Symposium on*

Microarchitecture (MICRO-30), Research Triangle Park, NC, December 1997.

[11] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, 1994.

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

[13] D. Chen. *Improving software pipelining with unroll-and-jam and memory-reuse analysis*. Michigan Technological University, Hougton, Michigan, master thesis edition, 1996.

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

[15] R. Ge. *Predicting the effects of register allocation on software pipelined loops*. Michigan Technological University, Hougton, Michigan, master thesis edition, 2002.

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

[17] D. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley and Sons, New York, NY, 1978.

[18] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. *the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[19] E. Rothberg M. Lam and M.E. Wolf. The cache performance and optimization of blocked algorithms. In *the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[20] Y. Ma, S. Carr, and R. Ge. Low-cost register-pressure prediction for scalar replacement using pseudo-schedules. In *33rd International Conference on Parallel Processing, (ICPP2004)*, pages 116–124, Montreal, Canada, 2004.

[21] B. Ramakrishna 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.

[22] S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. *The Journal of Instruction-level Parallelism (JILP)*, 2005.

[23] M. E. Wolf, D. E. Maydan, and D. K. Chen. Combining loop transformations considering caches and scheduling. In *the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286, Paris, France, 1996.