UC Irvine
ICS Technical Reports

Title
Optimal register allocation and assignment for loops

Permalink
https://escholarship.org/uc/item/81k8c2ws

Authors
Kolson, David J.
Nicolau, Alexandru
Dutt, Nikil

Publication Date
1995-04-01

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

SLBAR
Z
699
C3
no. 95-18

Optimal Register Allocation and Assignment for Loops*

David J. Kolson Alexandru Nicolau Nikil Dutt
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425

Technical Report #95-18

1 April 1995

Abstract

This paper presents a new technique for the problem of allocating and assigning registers to variables in loops. Traditionally, *cyclic variables* (variables written in the current iteration and read in subsequent iterations) are split at the loop boundary and treated as separate variables during register allocation and assignment. When these split variables are not assigned to the same register, register copy operations are necessary to match the register usages at the beginning and end of a loop iteration. Register copy operations, which are inherently overhead operations, have an adverse impact on the quality of the final design both in area (extra hardware—registers, busses—may be necessary) and in performance (register copy operations lengthen the schedule). Therefore, it is desirable to eliminate these spurious copy operations. In this paper, we describe a novel technique that incorporates loop unrolling into an assignment algorithm so that cyclic variables are used directly in subsequent iterations without requiring additional register copy operations, and also without requiring more registers than that used by the left-edge algorithm. We conducted experiments on some core numerical and image algorithms and observed optimal allocation.

*This work supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

1 Introduction

Allocation of hardware elements and the mapping or *binding* of these elements to behavioral objects are basic tasks in High-Level Synthesis [4, 5, 24]. Typically, a desired characteristic of the design solution is the minimum amount of hardware that will achieve given performance constraints. One problem in allocation and binding is to determine the minimal number of registers which are necessary to store values across states as well as the mapping of variables (or values[1]) to those registers.

The cyclic nature of loops considerably complicates this mapping process when a loop creates values in the current iteration that are used in future or subsequent iterations. The fundamental problem in handling loops with these cyclic (or loop-carried) variables is the matching of the variable-to-register mappings at the beginning and end of a loop. That is, the assignment of variables to registers at the beginning of the loop and at the end must match so that it is correct to iterate over the schedule. The traditional approach to overcoming this problem is to (arbitrarily) split any cyclic variable at the loop iteration boundary into two new variables which are then subjected to the mapping process.

If these two variables are not mapped to the same register, then register copy operations are necessary to make the register usages at the beginning and end of an iteration match. Various strategies exist (as discussed in the next section) to reduce the number of copy operations, but, in doing so, typically increase the number of registers (and connections) in the design and thus increase the area cost.

Reducing (or, ideally, completely eliminating) these register copy operations is important since they represent adversely impact the resulting design. To implement the copy operations, extra hardware (e.g., busses and/or temporary registers) may be necessary to provide the needed connections between the registers under consideration, thus increasing the area cost. Even if the necessary connections are present (i.e., existing data-paths can be utilized), copy operations lengthen the schedule, and thus, impact performance, especially since these overhead operations are contained within a looping construct.

In this paper we present a technique which maps variables to registers for behaviors with loops, such that *no* register copy operations are necessary to match register usages at loop beginning and end and *no more* registers than the maximum number of overlapping lifetimes are used (i.e., the same number of registers used by the left-edge algorithm). Our technique accomplishes this by incorporating loop unrolling into a register assignment algorithm. In contrast to other approaches, our algorithm produces an assignment of variables to registers which may possibly span multiple iterations of the original loop. Register copy operations are then unnecessary to match usages in subsequent iterations since those iterations have taken the previous iteration's assignment into account (i.e., values produced earlier are being used directly from their previously assigned registers).

This paper is organized as follows. In Section 2 we discuss previous work. In Section 3 we demonstrate the

[1] Without loss of generality, we use the terms *variable* and *value* synonymously in this paper.

deficiency of previous techniques in adequately removing spurious register transfers. In Section 4 we present our technique and in Section 5 we relate experiments that we conducted and our observed results. Finally in Section 6 we conclude.

2 Related Work

In High-Level Synthesis the problem of register assignment traditionally refers to determining the number of registers necessary to save values between time-steps. In the REAL project [11], the left-edge algorithm used in channel routing is adapted to the allocation of variable lifetimes and results in an optimal register allocation for basic blocks. Other approaches similar to REAL are used in SPLICER [14] and other synthesis systems [3, 17]. Another approach is based upon modelling non-overlapping variable lifetimes as cliques and applying a clique partitioning algorithm to the resulting graph [16, 23]. In the CADDY system [9], the allocation problem is formulated as a graph coloring problem. Also, in [7] a bipartite graph formulation is used.

In order to reduce the interconnect and multiplexer cost of scattered registers, some researchers have focused on grouping registers into memory modules [1, 2, 8, 13]. Also, [19] considers the allocation of *array* variables to memory modules.

However, these previous techniques do not satisfactorily handle register assignment when cyclic or loop-carried variables are present. Some work [6, 15, 20] has been done to improve these techniques for loops. These approaches (arbitrarily) break a cyclic variable's lifetime at loop boundaries, creating two "coupled" variables which the assignment process tries to assign to the same register. If the coupled variables are not assigned to the same register, register copy operations are necessarily inserted at the end of the loop to correctly set-up the next iteration.

In [6], a heuristic is used to try and fill the gap between the coupled variables and then the left-edge algorithm is applied. In [21], an initial assignment produced by the left-edge algorithm is iteratively improved by "permuting" the variables in registers at the end of the loop to obtain an allocation where register usages match at loop top and bottom. If it is impossible to permute the values, then registers are added to the design. In [15], an algorithm is presented which tries to keep coupled variables in the same register by modelling the assignment process by maximizing the overlapping of sets of intervals.

With all of these approaches, there exist situations in which it is impossible to assign registers without additional copy operations for a given connectivity, or that only permit removal of additional copy operations at the expense of extra hardware (i.e., more registers and connections). Section 3 presents such a case for illustration.

Our approach differs from previous work in that we incorporate loop unrolling into the register assignment algorithm. With our approach, registers are assigned to variables for a given iteration. Then, rather than adding

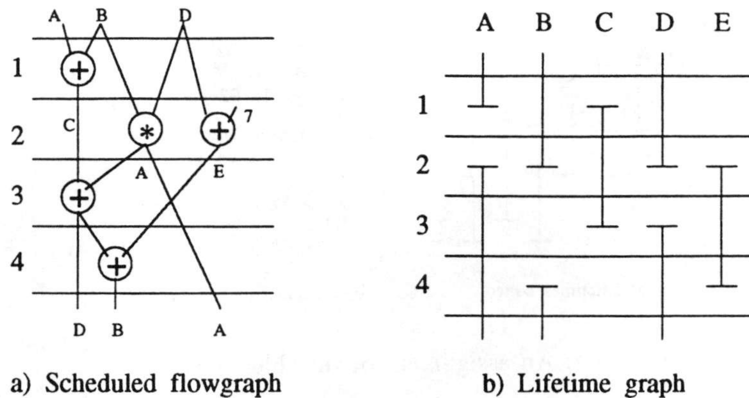


Figure 1: An example flowgraph and its corresponding variable lifetime graph.

copy operations to match usages, the loop is unrolled an iteration and assignment to the new iteration begins with the usages found at the end of the previous iteration. This process continues until a match is found in the register usages at the beginning and end of the unrolled loop. Thus, the register usages *naturally* match at the beginning and end of the loop since the assignment algorithm has produced a mapping which spans multiple iterations. Also, our assignment does not use more registers than that produced with the left-edge algorithm, since variables are not "forced" into particular register assignments.

3 A Motivating Example

In this section we present an example of a loop body where we assign registers by the traditional approach of splitting the cyclic variables at the loop boundary and apply the left-edge algorithm to allocate registers. Next, we demonstrate application of previous techniques on the example which results in additional register copy operations. Finally, we show the assignment using our technique that eliminates these copy operations.

3.1 An example

Figure 1 shows the loop body of a sample behavior represented as a dataflow graph and is scheduled into four cycles with the resource constraints of one adder and one multiplier, each with unit cycle latency. Also pictured is the corresponding variable lifetime graph. The execution model used here assumes that the target design has disjoint register read and write times. Therefore, for clarity in the lifetime graph, if a variable has its last use in a cycle, it dies at the cycle's mid-point. Conversely, if a variable is newly defined, it is created at the cycle's mid-point. This makes cases where variables may share registers readily apparent. For instance, it is clear that variables B and E may share a register in Figure 1(b).

Examining the variable lifetime graph, the minimal number of registers necessary to carry values across states

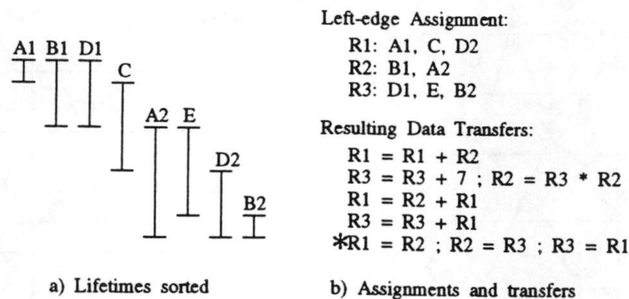


Figure 2: An assignment of variables to registers.

is three, since no more than three distinct variable edges cross a state boundary in Figure 1(a). Also, there are three cyclic variables: A, B and D. Existing techniques for allocating variables to registers would break these variables at the iteration boundary. This forms the variables $A1$, $A2$, $B1$, $B2$, $D1$, and $D2$, where a '1' denotes the loop entry segment of a variable's lifetime and a '2' denotes the loop exit segment. Figure 2(a) depicts the split variables sorted by birth times. Applying the left-edge algorithm to these variables gives the register assignment and the resulting loop body show in Figure 2(b) (transfers separated by semi-colons are executed concurrently). Note that the last step of the loop body in Figure 2(b) requires three additional register copy (move) operations; the variables in the registers must be completely permuted to correctly set-up those variables for the next iteration (i.e., move $D2$ to **R3**, $A2$ to **R1** and $B2$ to **R2**). This represents an overhead which cannot be eliminated without exploiting loop unrolling or additional registers.

3.2 Stok's Approach

Stok's approach [21] to removing register copy operations at the end of loops is to start with an initial assignment (produced by the left-edge algorithm) and then to iterate over a "permute" phase (which is formulated as a multi-commodity flow problem) and an allocate phase (which increases the number of registers by one each time the permute phase fails) until no transfers are necessary.

Figure 3(a) graphically indicates the overlapping of variable lifetimes across cycles from the initial left-edge assignment found in Figure 2. The original flowgraph is scheduled into four cycles, therefore the line labelled "t5" represents the first cycle of the next iteration.

Recall that three register copy operations are required to correctly set-up the values for the next iteration. After applying Stok's algorithm to the initial assignment the values are re-arranged in the registers so that one less register copy operation is necessary. This resulting assignment is shown in Figure 3(b) and requires two register copy operations to swap the values in **R1** and **R3**. However, since it is not possible to further reduce the number of required copy operations (i.e., the variables cannot be permuted around to obtain an assignment which contains no copy operations), the next iteration of Stok's algorithm adds a fourth register to the design

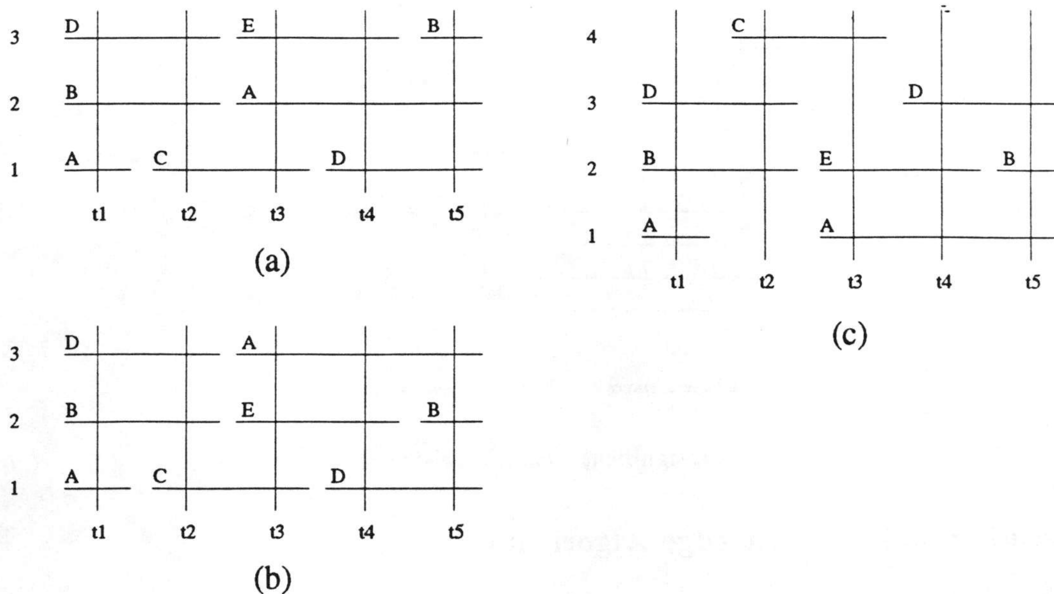


Figure 3: Applying Stok's algorithm to example flowgraph.

and the permute phase is repeated. The addition of the fourth register results in a mapping of variables to registers with no copy operations as depicted in Figure 3(c). Thus, using Stok's approach, extra hardware (i.e., another register and its associated connections) is necessary to remove all copy operations.

3.3 Our Solution to the Example

As we have noted, it is not possible to derive an assignment of the variables to registers for this example such that the cyclic variables are in the same registers at the loop beginning and end. Rather than adding register copy operations at the loop end or adding a register to the design, we unroll the loop for another iteration and assign registers to the new iteration based upon the assignment found at the end of the previous iteration.

Figure 4 shows the example unrolled for another iteration (i.e., one execution of the new loop corresponds to two iterations of the original loop). At the beginning of the loop the variables A, B and D are in registers **R1**, **R2** and **R3**, respectively. At the end of the first iteration, the variables A, B and D have been mapped to registers **R3**, **R2** and **R1**, respectively. This is the assignment that is used as the starting point for assigning registers in the next iteration. At the end of the second iteration, the variables A, B and D have been mapped to **R1**, **R2** and **R3**, respectively, which ***naturally*** (i.e., as a result of the dataflow and register assignment, and without copy operations) matches the assignment found at the beginning of the first iteration. (In Section 4, we show exactly how this solution is obtained.) Therefore, it is possible to iterate over this new assignment without introducing register copy operations and without adding more registers.

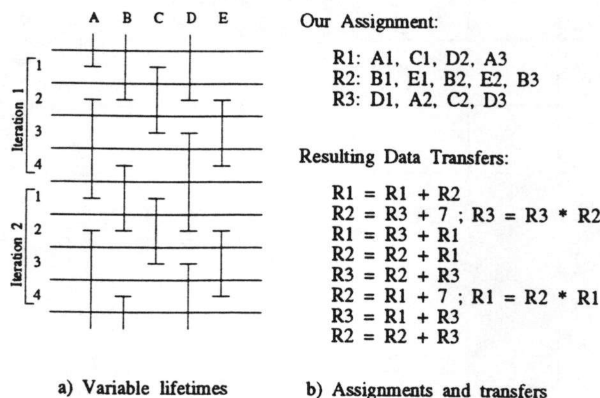


Figure 4: An assignment with no register copy operations.

3.4 Unrolling and the Left-edge Algorithm

It is important to note that the solution shown above is not derived simply from unrolling the loop one iteration and applying the left-edge algorithm. In Figure 5, the example has been unrolled one iteration and the corresponding variable lifetimes are sorted by birth times (with the cyclic variables split at the final loop boundary). Applying the left-edge algorithm to the sorted lifetimes gives the assignment also pictured. Although this strategy has eliminated one register copy operation (by matching the variable A in register **R1** at the loop beginning and end), it did not result in an allocation without register copy operations—the variables in **R2** and **R3** must be swapped.

As a matter of fact, this is not a matter of unrolling for "enough" iterations. In Figure 6, the example has been unrolled for three iterations and the variables lifetimes are sorted by birth times (again, with the cyclic variables split at the loop boundaries). After applying the left-edge algorithm to these variables, the resulting assignment requires three register copy operations. This mismatching is due to the fundamental problem with the approach of (arbitrarily) splitting variables at the loop boundary. It creates two variables that must be matched to avoid register copy operations, rather than modifying the register transfers to used the variables from their new locations.

4 Our Technique

In the preceding example, it is not obvious why only two iterations suffice to produce a mapping of variables to registers such that no register copy operations are necessary. In general, it may be necessary to unroll the loop for more iterations to produce such a mapping.

In this section we present our algorithm for assigning registers to variables in loops. First, we discuss our model for variable accesses and present *variable tracking* which is a simple method of assigning variables to

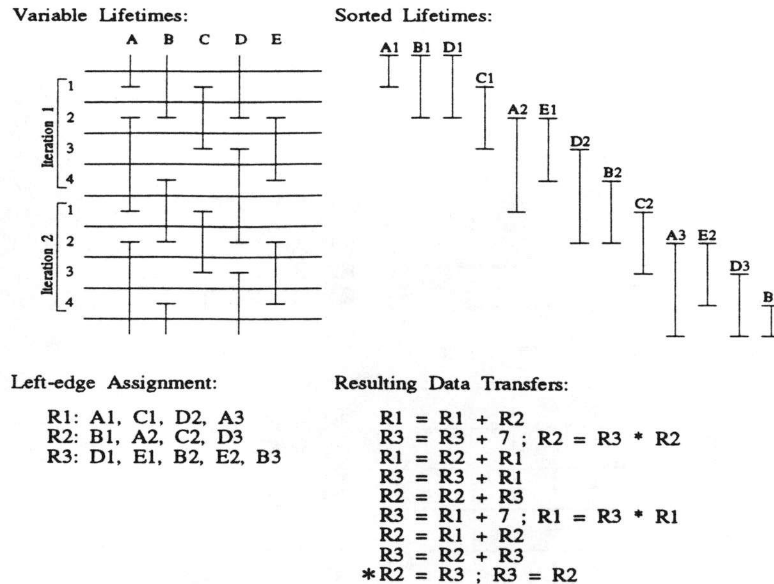


Figure 5: Unrolling and applying the left-edge algorithm.

registers. Then, we present an optimal, but exponential, algorithm and a heuristic modification that achieves equally good results on benchmarks.

4.1 Variable Access Model

We model the accessing of variables in each state with a *variable access stream.* This stream indicates which variables are read, which are written and which become dead. A read of a variable is simply denoted by the variable (e.g., A), a write is denoted by an asterisk following a variable (e.g., $A*$) and the last use of a variable is denoted by a minus sign following a variable (e.g., $A-$). Also, because we assume that the register read and write times in a cycle are disjoint, all reads in a cycle are found before any of the writes in the variable access stream. Parenthesis are used to group all variable reads and writes occuring in a particular state.

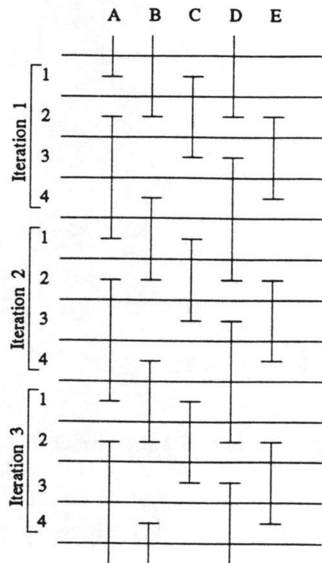
As an example, the variable access streams for each state of the earlier example (found in Figure 1) are shown in Figure 7. In the first state, the variables A and B are read (with A having its last use) and the variable C is defined[2]. Therefore, the variable access stream is "$(A - BC*)$". Once all of the streams for each state are derived, they are concatenated to form the variable access stream for the loop which is also shown in Figure 7.

4.2 Variable Tracking

Variable tracking is a simple mechanism for keeping track of which register contains particular variable and denotes the mapping of variables to registers found at state boundaries. A *variable tracking graph* is a graph

[2] A *definition* of a variable is synonymous with a write to a variable and updates the value of that variable.

Variable Lifetimes:



Left-edge Assignment:

R1: A1, C1, D2, A3, C3, D4
R2: B1, A2, C2, D3, A4
R3: D1, E1, B2, E2, B3, E3, B4

Resulting Data Transfers:

R1 = R1 + R2
R3 = R3 + 7 ; R2 = R3 * R2
R1 = R2 + R1
R3 = R3 + R1
R2 = R2 + R3
R3 = R1 + 7 ; R1 = R3 * R1
R2 = R2 + R1
R3 = R3 + R2
R1 = R1 + R3
R3 = R2 + 7 ; R2 = R2 * R3
R1 = R1 + R2
R3 = R1 + R3
*R1 = R2 ; R2 = R3 ; R3 = R1

Sorted Lifetimes:

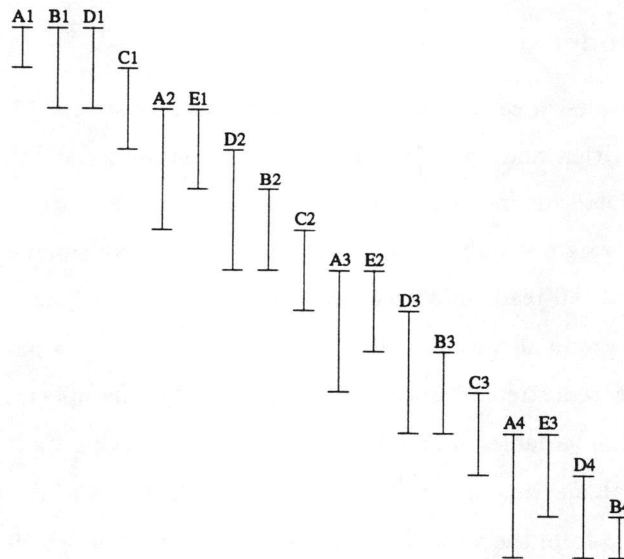


Figure 6: Unrolling the example three iterations.

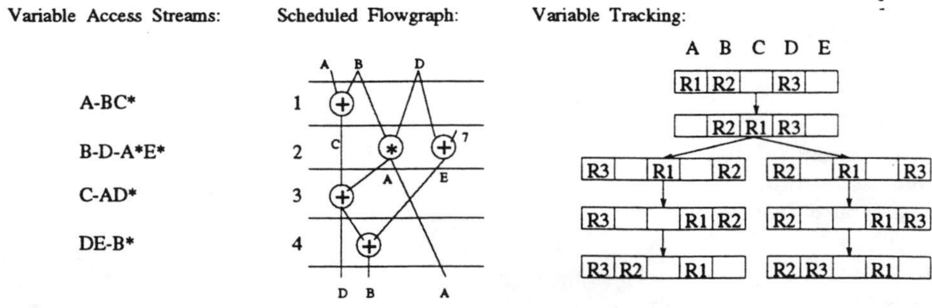


Figure 7: The example and its variable access streams.

where each node corresponds to a particular mapping of variables to registers and the edges between nodes represent the variable access streams which were executed in that state. To derive the mapping for the next state given the mapping for the current state and its variable access stream, the registers belonging to variables that die (if any) are de-allocated and then registers are (re-)assigned to those variables which are defined.

Figure 7 shows the variable maps and the tracking mechanism for the example presented earlier. The initial mapping to the loop is: $\{ A \rightarrow \mathbf{R1}, B \rightarrow \mathbf{R2}, D \rightarrow \mathbf{R3} \}$. This is the mapping found at the beginning of state 1. To derive the next mapping, the variable access stream, $A - BC*$, is considered. Since the variable A dies, the register **R1** is de-allocated and then subsequently re-assigned to the variable C giving the mapping found at the boundary between states 1 and 2.

Notice that in state 2, two variables (B and D) have their last use and two variables (A and E) are created. In this case, to derive an optimal solution, we explore all possibilities. That is, since B and D die and A and E are created, two possible mappings—assigning $B's$ register to A and $D's$ register to E or assigning $B's$ register to E and $D's$ register to A—are generated. Register assignment then continues with both of these mappings.

4.3 An Optimal Algorithm

Our approach is to iteratively unroll the loop for one iteration and to track (assign) variables over that new iteration, based upon the mappings found at the end of the previous iteration. Then, the variable mappings found at the end of the loop are checked against the variable mappings found at previous iteration beginnings to detect if there is a match. If so, then an assignment of variables to registers has been found which requires no register copy operations. If no matches are found, then the process of unrolling and assignment is repeated.

In order to ensure optimality (i.e., an assignment which spans the minimal number of iterations), whenever variables are defined, multiple nodes are constructed in the tracking graph. These nodes correspond to assigning each free register to a defined variable. Once the variables have been tracked, assignment continues with each

```
Procedure Optimal-Assign ( Init : Initial register assignment;
                           VA : Variable access pattern;
                           N : number of registers)
Begin
   Set allocation_found to false
   Forany registers ∉ Init
      Add registers to free_regs
   Loop
      Foreach state in VA
         Foreach map in curr_maps
            Add all last-use registers to free_regs
            Generate all permutations of defined vars. in free_regs
            Foreach permutation
               Add variables in map which are live
         If (state is the loop end)
            Foreach map in var_maps
               If (current mapping matches map)
                  Set assignment_found to true
                  Set assignment to map
               Endif
            End
            If (assignment_found is false)
               Unroll the loop one iteration
            Else
               Add current mapping to var_maps
            Endif
         Endif
   Until assignment_found
   Return assignment
End Heuristic-Assign
```

Figure 8: An optimal register assignment algorithm.

of the nodes generated for that state. This leads to an exponential, but optimal, algorithm since all possibilities for assigning variables to registers have been tried.

Our algorithm for optimally assigning variables to registers is shown in Figure 8. The algorithm takes as input the variable access stream for the loop, an initial mapping of variables to registers and the number of registers (both of which can be found by applying the left-edge algorithm to the loop). After some initialization, our algorithm iterates over an assignment phase, where the variables are tracked, and an unrolling phase (if an assignment to the loop is not found).

Figure 9 demonstrates how our algorithm operates on the example presented earlier. The initial assignment of variables to registers is { $A \rightarrow$ **R1**, $B \rightarrow$ **R2** and $D \rightarrow$ **R3** }. As illustrated before, in state 2, two variables, A and E are defined and two possible mappings result. Continuing assignment with both of these mapping, at the end of the first iteration the mappings { $A \rightarrow$ **R3**, $B \rightarrow$ **R2**, $D \rightarrow$ **R1** } and { $A \rightarrow$ **R2**, $B \rightarrow$ **R3**,

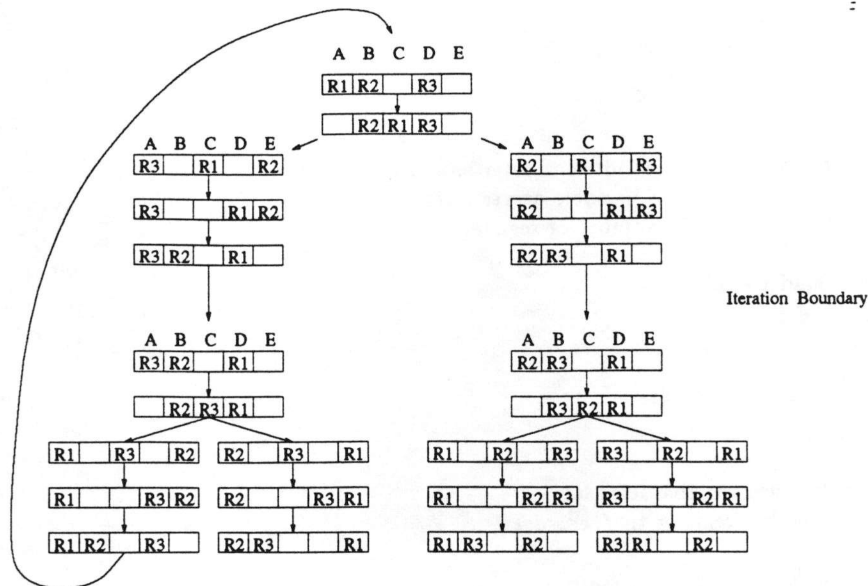


Figure 9: Applying the optimal algorithm to the example.

$D \rightarrow$ **R1** } result. Since neither of these mappings match any previous mapping found at the beginning of an iteration (which is one in this case), the loop is unrolled for an iteration and variables are assigned to the new iteration.

Each of the mappings from iteration one is used in assigning registers to the second iteration. Again, when registers are assigned in state 2, all possibilities are tried. At the end of iteration two, the resulting mappings are checked against the (three) previous mappings occuring at an iteration beginning. Since a match is found which produces a cycle (i.e., it is possible to iterate over that assignment), the process terminates and the resulting assignment spans two iterations.

4.4 A Heuristic Algorithm

The complexity of the optimal algorithm arises from trying all possibilities of variables in registers when a variable is defined. To reduce the search space, we employ a heuristic which simply keeps track of the last register assigned to each variable. Then, when newly defined variables are assigned to registers, we check to see if that variable's last assigned register is free. If so, then it is assigned, otherwise another register is assigned and the register information is updated[3]. Figure 10 contains a heuristic version of our algorithm that implements this strategy.

Our heuristic version of the optimal algorithm still retains the property that the final solution has no register copy operations. However, the solution derived by our heuristic may span more iterations than the minimal

[3] To avoid ordering problems, the heuristic is consulted for each variable before any registers are assigned.

```
Procedure Heuristic-Assign ( Init : Initial register assignment;
                             VA : Variable access stream;
                             N : number of registers)
Begin
   Set allocation_found to false
   Foreach register ∉ Init
      Add register to free_regs
   Foreach variable
      Set last_reg(variable) to {ϕ}
   Loop
      Foreach state in VA
         Foreach last use variable in state
            Add variable's register to free_regs
         End
         Foreach defined variable in state
            If (last_reg(variable) ∈ free_regs )
               Assign variable to last_reg(variable)
               Remove from free_regs
            Endif
         End
         Foreach unassigned definition variable in state
            Assign a free register to variable
            Remove from free_regs
            Update last_reg(variable)
         End
         If (state is the loop end)
            Foreach map in var_maps
               If (current mapping matches map)
                  Set assignment_found to true
                  Set assignment to map
               Else
                  Add current map to var_maps
               Endif
            End
            If (assignment_found is false)
               Unroll the loop one iteration
            Endif
         Endif
   Until assignment_found
   Return assignment
End Heuristic-Assign
```

Figure 10: A heuristic version of the assignment algorithm.

Variable Tracking Information:

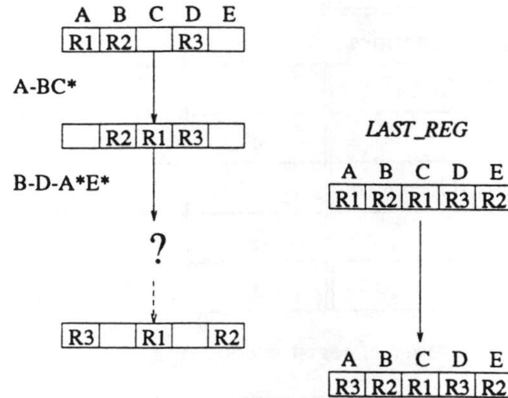


Figure 11: Applying our heuristic to the example.

number produced by the previous algorithm.

As an example, Figure 11 shows the assignment process for our example using the heuristic version of our algorithm. At this point, we are considering assigning registers to variables A and E. Also, the two registers assigned to B and D become free. In the optimal algorithm, we would generate two mappings: one where { $A \rightarrow$ R2, $E \rightarrow$ R3 } and vice versa. However, for the heuristic version only one mapping is generated by consulting the *LAST_REGS* of A and E. The last register assigned to A (**R1**) is not free, but the last register assigned to E (**R2**) is. Therefore, **R2** is assigned to E and **R3** is assigned to A and the last assigned register information for A is updated.

5 Experiments and Results

In this section we describe the experiments that we conducted with our algorithm and the results that we observed. We have implemented both optimal and heuristic versions of our algorithm and have applied them to a suite of benchmarks consisting of six numerical and image processing behaviors. The two-dimensional hydrodynamics benchmark is adapted from [22] and the inner product benchmark is adapted from [10]. The wavelet and predictor-corrector image compression benchmarks are adapted from [18] and the Laplace and low-pass filters are adapted from [12].

Schedules for each benchmark were generated with the resource constraints of one adder with one-cycle latency and one multiplier with one-cycle latency. The variable lifetimes were derived from these schedules and used to conduct a series of experiments.

The first experiment was designed to study the number of register copy operations necessary at the loop

Benchmark	Left-edge	Our technique	
		Optimal	Heuristic
2D-Hydrodynamics	4	0	0
Inner Product	2	0	0
Wavelet	3	0	0
Predictor-Corrector	4	0	0
Laplace	9	0	0
Low Pass	8	0	0

Table 1: Number of register copy operations in schedules.

Benchmark	Optimal	Heuristic
2D-Hydrodynamics	2	2
Inner Product	3	4
Wavelet	2	2
Predictor-Corrector	2	2
Laplace	3	3
Low Pass	3	4

Table 2: Number of loop iterations spanned by register assignment.

end when using the traditional approach where cyclic variable lifetimes are split at the loop boundary. For this experiment we took the variable lifetimes, split them at the loop boundaries and applied the left-edge algorithm. The resulting assignment was examined and the number of register copy operations necessary at the end of the loop was noted. These results are found in column one of Table 1 which is labelled "Left-edge."

The variable assignments at loop end as well as the number of registers found in the mapping produced by the left-edge algorithm was used as input to our algorithm. The variable access streams were derived from the schedules and also input to our algorithm. Table 1 contains the number of register copy operations found in the assignments produced by the optimal and heuristic algorithms. In all cases, the results are zero, that is, no register copy operations are necessary since our assignments naturally match the register usages at loop beginning and end.

Two concerns follow from this: *How many iterations did our solutions span?* and *What is the difference in performance between schedules with copy operations and schedules without?*. Experiments two and three were designed to answer these questions, respectively.

In experiment two, we noted the number of iterations that the variable assignments produced by our algorithm

Benchmark	Left-edge	Our technique	% Improvement
2D-Hydrodynamics	15	13	13%
Inner Product	13	10	23%
Wavelet	20	17	15%
Predictor-Corrector	14	12	14%
Laplace	23	19	17%
Low Pass	15	12	20%

Table 3: Number of cycles in schedules.

spanned. These results are found in Table 2. In most cases, our heuristic version derived a solution that spanned the same number of iterations as the optimal. For all of the solutions, the number of iterations spanned by the optimal and heuristic assignments is small enough so as not to be prohibitive in terms of the resulting code size.

In experiment three, we studied the effects of register copy operations on performance. Using the assignments produced by the left-edge algorithm and by our algorithms, registers were assigned to the schedules. Then, for those schedules which were assigned by the left-edge algorithm, necessary register copy operations were added. For the purposes of scheduling, it is assumed that the existing connectivities were used to perform the register copy operations (i.e., the adder and multiplier were used, incurring a one-cycle latency for executing the copy operations). The number of cycles in those schedules was counted and these results appear in Table 3.

Because both our optimal and heuristic versions derive assignments which do not contain register copy operations, only one column is used for these results in Table 3. Also, the results for our assignments are normalized to one iteration. That is, since our assignments span multiple iterations, the total number of cycles in the schedules was divided by the number of iterations spanned by the register assignment.

In the column labelled "% Improvement" we note the percentage improvement of the schedules produced by our technique over those produced by the traditional approach. For the scientific benchmarks, we observed improvements of 13% and 23% and, for the image benchmarks, we observed improvements in performance between 15% and 20%. This significant performance improvement clearly demonstrates the utility of our register assignment technique.

6 Conclusion

In this paper we have presented a novel algorithm which assigns variables to registers in the presence of loops. Traditional techniques arbitrarily break variables whose lifetimes cross iteration boundaries at those boundaries. Then, those "de-coupled" variables are assigned to registers. When the de-coupled variables are not assigned to the same register, register copy operations are necessary to correctly set-up the variables for subsequent

loop iterations. However, those register copy operations have an impact on the design, both on the area—new connections and/or hardware may be necessary—and on the performance—the copy operations increase the length of the schedule.

Our technique incorporates loop unrolling into an assignment algorithm so that cyclic variables assigned to a particular register are subsequently used directly from that register. In this way, no register copy operations are necessary to move variables around—the schedule already has been modified to use variables directly from the register previously assigned. Also, our algorithm uses no more registers than that used by the left-edge algorithm. That is, only enough registers as the maximal simultaneously live variables is necessary with our approach. We have conducted experiments on some core numerical and image algorithms and have observed improvements of between 13% to 23% on these benchmarks.

References

[1] I. Ahmed and C. Y. R. Chen. Post-processor for Datapath Synthesis Using Multiport Memories. *IEEE International Conference on Computer-Aided Design '91*, 1991.

[2] M. Balakrishnan, A. K. Majumdar, D. K. Banerji, J. G. Linders, and J. C. Majithia. Allocation of Multiport Memories in Data Path Synthesis. *IEEE Transactions on Computer-Aided Design*, 7(4), April 1988.

[3] R. A. Bergamaschi, R. Camposano, and M. Payer. Allocation Algorithms Based on Path Analysis. *Integration, the VLSI journal*, 13, 1992.

[4] R. Camposano and W. Wolf. *High Level VLSI Synthesis*. Kluwer Academic Publishers. Norwell, MA., 1991.

[5] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers. Norwell, MA., 1992.

[6] G. Goossens. *Optimization Techniques for Automated Synthesis of Application-specific Signal Processing Architectures*. PhD thesis, KU Leuven, 1989.

[7] C.-Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu. Data Path Allocation Based on Bipartite Weighted Matching. *27th DAC*, 1990.

[8] T. Kim and C. L. Liu. Utilization of Multiport Memories in Data Path Synthesis. *30th DAC*, 1993.

[9] H. Krämer and W. Rosenstiel. System Synthesis Using Behavioural Descriptions. *1st EDAC*, 1990.

[10] D. J. Kuck. *The Structure of Computers and Computations*, volume 1. Wiley & Sons, 1978.

[11] F. J. Kurdahi and A. C. Parker. REAL: A Program for Register Allocation. *24th ACM/IEEE Design Automation Conference*, 1987.

[12] J. S. Lim. *Two-Dimensional Signal and Image Processing*. Prentice Hall Signal Processing Series, 1990.

[13] P. Marwedel. The MIMOLA Design System: Tools for the Design of Digital Processors. *DAC-84*, June 1984.

[14] B. M. Pangrle. Splicer: A Heuristic Approach to Connectivity Binding. *25th DAC*, 1988.

[15] C. Park, T. Kim, and C. L. Liu. Register Allocation for Data Flow Graphs with Conditional Branches and Loops. *Euro-DAC '93*, 1993.

[16] P. Paulin. Scheduling and Binding Algorithms for High-Level Synthesis. *26th DAC*, 1989.

[17] P. G. Paulin and J. P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Transactions on the Computer-Aided Design of Integrated Circuits and Systems*, 8(6), June 1989.

[18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.

[19] L. Ramachandran, D. D. Gajski, and V. Chaiyakul. An Algorithm for Array Variable Clustering. *EDAC '94*, 1994.

[20] L. Stok. Interconnect Optimisation During Data Path Allocation. *European Design Automation Conference (EDAC)*, 1990.

[21] L. Stok. *Architectural Synthesis and Optimization of Digital Systems*. PhD thesis, Eindhoven University of Technology, 1991.

[22] Y. Tanaka, K. Iwasawa, Y. Umetani, and S. Gotou. Compiling Techniques for First-Order Linear Recurrences on a Vector Computer. *Journal of Supercomputing*, 4(1), March 1990.

[23] C. J. Tseng and D. P. Siewiorek. Facet: A Procedure for the Automated Synthesis of Digital Systems. *20th DAC*, 1983.

[24] J. Vanhoof, K. Van Rompaey, I. Bolsens, G. Goossens, and H. De Man. *High Level Synthesis for Real Time Digital Signal Processing*. Kluwer Academic Publishers. Norwell, MA., 1993.