Minimization of Memory Traffic in High-Level Synthesis*

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

Abstract

This paper presents a new transformation for the scheduling of memory-access operations in High-Level Synthesis. This transformation is suited to memory-intensive applications with synthesized designs containing a secondary store accessed by explicit instructions. Such memory-intensive behaviors are commonly observed in video compression, image convolution, hydrodynamics and mechatronics. Our transformation removes load and store instructions which become redundant or unnecessary during the transformation of loops. The advantage of this reduction is the decrease of secondary memory bandwidth demands. This technique is implemented in our Percolation-Based Scheduler which we used to conduct experiments on core numerical benchmarks. Our results demonstrate a significant reduction in the number of memory operations and an increase in performance on these benchmarks.

1 Introduction

Traditionally, one of the goals in High-Level Synthesis (HLS) is the minimization of storage requirements for synthesized designs [5, 7, 15, 19]. As the focus of HLS shifts towards the synthesis of designs for inherently memory-intensive behaviors [6, 8, 14, 16, 18], memory optimization becomes crucial to obtaining acceptable performance. Examples of such behaviors are abundant in video compression, image convolution, speech recognition, hydrodynamics and mechatronics. The memory-intensive nature of these behaviors necessitates the use of a secondary store (e.g., a memory system), since a primary store (e.g., register storage) sufficiently large enough would be impractical. This memory is explicitly addressed in a synthesized system by memory operations containing indexing functions. However, due to bottlenecks in the access of memory systems, memory accessing operations must be effectively scheduled so as to improve performance.

Our strategy for optimizing memory access is to eliminate the redundancy found in memory interaction when scheduling memory operations. Such redundancies can be found within loop iterations, possibly over multiple paths, as well as across loop iterations. During loop pipelining [17, 13] redundancy is exhibited when values loaded from and/or stored to the memory in one iteration are loaded from and/or stored to the memory in future iterations. For example, consider the behavior:

for $i = 1$ to N
  for $j = 1$ to N
    $a[i] := a[i] + \frac{1}{2}(b[i][j-1] + b[i][j-2])$
    $b[i][j] := F(b[i][j])$
  end
end

The inner loop would normally require four load and two store instructions per iteration. However, after application of our transformation, the inner loop contains only one load and one store.

Previous work in reducing memory accessing balances load operations with computation [3]. However, their algorithm only removes redundant loads and only deals with single-dimensional arrays and single flow-of-control. In [6] a model for access dependencies is used to optimize memory system organization. In [16] background memory management is discussed, but no details of an algorithm are present. Therefore, it is not clear what approach is taken in determining redundancy removal, nor the general applicitivity of the technique. Related work includes the minimization of registers [8, 14], the minimization of the interconnection between secondary-store and functional units [11], and the assignment of arrays to storage [18].

Our transformation has many significant benefits. By eliminating unnecessary memory operations that occur on the critical dependency path through the code, the performance of the resulting schedule can increase dramatically: the length of the critical path can be shortened, thus generating more compact schedules and reducing code size. Also, due to our transformation's local nature, it integrates easily into other parallelizing transformations [4, 13]. Another benefit is the possible savings in hardware due to the decrease in memory bandwidth requirements and/or the exploration of more cost-effective implementations.

2 Program Model

In our model, a program is represented by a control data-flow graph where each node corresponds to operations performed in one time step and the edges between

*This work was supported in part by NSF grant CCR8704367, ONR grant N0001486K0215 and a UCI Faculty Research Grant.

nodes represent flow-of-control. Initially, each node contains only one operation. Parallelizing a program involves the compaction of multiple operations into one node (subject to resource availability).

Memory operations contain an indexing function, composed of a constant base and induction variables (iv's), and either a *destination* for load operations or a *value* for store operations[1]. The semantics of a load operation are that issuing a load reserves the *destination* (local storage) at issuance time (i.e., *destination* is unavailable during the load's latency). For the purpose of dependency analysis on memory operations, each contains a *symbolic expression* which is a string that formulates the indexing function without iv's. (During loop pipelining these expressions must be updated w.r.t. iv's.)

The *initial_analysis* algorithm in Fig. 1 computes initial program information. Detecting loop invariants and iv's and building iv use-def chains can be done with standard algorithms found in [1] and stored into a database. The function *build_symbolic_exprs* creates symbolic expressions for each memory operation in the program by getting the iv definitions that define the current operation's indexing function and deriving an expression for each. Next, the base of the memory structure is added to each expression. An operation is then annotated with its expression, combining multiple expressions into one of the form "((*expr1*) *or* ... *or* (*exprN*))."

The function *derive_expr* constructs the expression "(LoopId * Const)" if iv is self-referencing (e.g. $i = i + const$) where LoopId is the identifier of the loop over which iv inducts and Const is a constant derived from the constant in the iv operation multiplied by a data size and possibly other variables and constants[2]. In the introductory example, the data size for the j loop is the array element size and for the i loop is the size of a column (or row) of data. If iv is defined in terms of another iv (e.g. $i = j + 1$, where j is an iv) then recursive calls are made on all definitions of that other iv. In this case, marking of iv's is necessary to detect cyclic dependencies which are handled by a technique called *variable folding*. Essentially variable folding determines an initial value of a variable on input to the loop or resulting from the first iteration (i.e. values which are loop-carried are not considered) from the reverse-flow of the graph. The result can be a constant or another variable (which is recursively folded, until the beginning of the loop is reached).

Fig. 2 shows a sample behavior and its CDFG annotated with symbolic expresssions. The load from A builds the expression "((8 * L0) + (4 * L0))" which is the addition of 2 (the const for iv j) times 4 (the element size) and 1 (the const for iv i) times 4. The second loop over k adds the expression "(400 * L1)." Finally the base address of A is added. For the store operation, the expression "(12 * L0)" is created which is 1 times 4 times 3 (the constant in the behavior). Due to the +1 in the index expression, the constant 4 is added to the base address of A.

[1] We use the term **argument** to refer to *destination* if the operations is a load or to *value* if the operation is a store.

[2] For clarity we present a simplified algorithm. More complex analysis (based on [12]) has been implemented in our scheduler.

```
Procedure initial_analysis(program)
begin
  /* Detect loop invariants. */
  /* Detect induction variables in program. */
  /* Build iv use-def chains in program. */
  build_symbolic_exprs(program)
end initial_analysis

Procedure build_symbolic_exprs(program)
begin
  foreach mem_op in program
    /* Set iv_defs to the possible iv defs found in DB. */
    foreach iv_group in iv_defs
      new_expr = derive_expr(iv_group)
      /* Add Base of mem_op to new_expr. */
      /* Annotate mem_op with new_expr. */
    end
  end
end build_symbolic_exprs

function derive_expr(iv_group)
begin
  foreach iv in iv_group
    if (/* iv is marked */) then
      /* Do variable folding. */
    else if (/* iv is self-referencing */)
      /* Return the string "(Const * LoopId))" */
    else
      /* Mark iv, then recursively derive */
      /* the iv that defines this iv. */
    end if
  end
end derive_expr
```

Figure 1: Initial program analysis.

3 Memory Disambiguation

Memory disambiguation is the ability to determine if two memory access instructions are *aliases* for the same location [1]. In our context, we are interested in static memory disambiguation, or the ability to disambiguate memory references during scheduling. In the general case, memory indexing functions can be arbitrarily complex due to explicit and implicit induction variables and loop index increments. Therefore, a simplistic pattern matching approach to matching loads and stores over loop iterations cannot provide the power of memory aliasing analysis. For instance, in the following behavior, if arrays a and b are aliases:

for $i = 1$ to N
  $a[i] := \frac{1}{2}b[i-1] + \frac{1}{3}a[i-2]$
  $Coef[i] := b[i] + 1$
end

pattern matching will fail to find the redundancy.

In our scheduler, memory disambiguation is based on the well-known *greatest common divisor*, or GCD test [2].

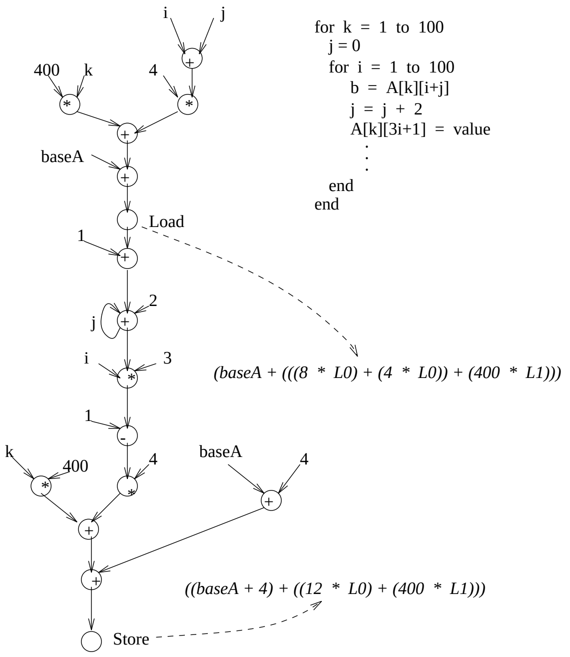

Figure 2: Symbolic expressions example.

Performing memory disambiguation on two operations, *op1* and *op2*, involves determining if the difference equation: *(op1's symbolic expression) - (op2's symbolic expression) = 0* has any integer solution. Fig. 3 contains an algorithm to disambiguate two memory references.

This algorithm works by iterating over all expressions of operations one and two, thereby testing each possible address that the two operations can have. The first step in disambiguating two expressions is to convert them into the sum of products form "((a * b)+...+(y * z))." Next, operation two's expression is subtracted from operation one's. If the resultant expression is not linear then the disambiguator returns CANT_TELL, otherwise the gcd of coefficients of the equation is solved for. If the gcd does not divide all terms, there is no dependence between *op1* and *op2*.

Returning to the example in Fig. 2, if the load from iteration *i + 1* is overlapped with the store from iteration *i*, the disambiguator determines that the updated expression for the load minus the store's expression is 0, exposing the redundancy in loading a value which has just been computed.

If the disambiguator cannot determine that two memory operations refer to the same location, we follow the conservative approach that there is a dependence between them (i.e., no optimization can be done). Assertions (source-level statements such as certain arrays reside in disjoint memory space, absolute bounds on loops, etc.) can be used to allay this. Also, providing the user with the information the dismabiguator has derived and querying for a result to the dependence question is an alternate, interactive approach.

```
function disambiguate(op1, op2)
begin
  foreach ex1 in op1's expressions
    foreach ex2 in op2's expressions
      /* Convert ex1 and ex1 into sum of products form. */
      /* Set expr to ex1 - ex2. */
      /* If expr is not linear, return CANT_TELL. */
      /* Solve GCD of coefficients of expr. */
      /* If sol and divides all terms return EQUAL */
      /* else return NOT_EQUAL. */
    end
  end
end disambiguate
```

Figure 3: Disambiguating memory references.

```
function redundant_elimination(op, from_step)
  begin
    if (INVARIANT(op)) then
      status = remove_inv_mem_op(op, from_step)
      /* if op was removed, return REMOVED */
    foreach memory operation, mem_op, in to
      if (disambiguate(op, mem_op) == EQUAL) then
        switch op-mem_op
          case load-load:
            return do_load_load_opt(op, mem_op)
          case load-store:
            status = try_load_store_opt(op, mem_op)
            /* if op was removed, return REMOVED */
          case store-store:
            /* If op's arg and mem_op's arg have */
            /* the same reaching defs, delete op, */
            /* and update necessary information. */
            return REMOVED
          case store-load:
            return ANTI-DEPENDENCE
        end
      end if
    end
  end redundant_elimination
```

Figure 4: Redundant elimination algorithm.

4 Reducing Memory Traffic

Our solution to reducing the amount of memory traffic in HLS is to make explicit the redundancy in memory interaction within the behavior and eliminate those extraneous operations. Our technique is employed during scheduling rather than as a pre-pass or post-pass phase; a pre-pass phase may not remove all redundancy since other optimizations can create opportunities that may not have otherwise existed while a post-pass phase cannot derive as compact a schedule since operations eliminated on the critical path allow further schedule refinement.

4.1 Algorithm in Detail

Fig. 4 shows the main algorithm for removing unnecessary memory operations. This function is invoked in our Percolation-based scheduler [17] by the *move_op* transform (or any suitable local code motion routine in other systems) when moving a memory operation into a previous step that contains other memory operations.

The function *redundant_elimination* checks to see if the memory operation is invariant. If so, then the function *remove_inv_mem_op* tries to remove it. If it is not invariant or could not be removed, then *op* is checked against each memory operation in the previous step for possible optimization. If two operations refer to the same location then the appropriate action is taken depending upon their types. The load-after-load and load-after-store cases will be discussed shortly. In the case of a store-after-store, the first operation is dead and can be removed if it stores the same argument as the second and the argument has the same reaching definitions. We choose to simply remove *op*, rather than removing *mem_op* and moving op into its place. For the store-after-load, nothing is done as this is a false (anti-) dependency that should be preserved for correctness. Status reflecting the outcome is returned, allowing operations to continue to move if no redundancy was found.

4.1.1 Removing Invariants

Removing invariant memory operations is slightly different from general loop invariant removal. Traditional loop invariant removal moves an invariant into a pre-loop time step. For load operations this is correct; for store operations it is not. Conceptually, invariant loads are "inputs" to the loop, while invariant stores are "outputs." Therefore, loads must be placed in pre-loop steps and stores must be placed in loop exit steps.

An algorithm to perform invariant removal appears in Fig. 5. The conditions necessary for loop invariant removal (adapted from [1]) are: 1) the step that op is in must dominate all loop exits (i.e., op *must* be executed every iteration), 2) only one definition of the variable (for loads) or memory location (for stores) occurs in the loop and 3) no other definition of the variable or memory location reaches their users. Additionally, store operations require that the definition of its argument be the same at the loop exits so that correctness is preserved. If these conditions are met, then the operation can be hoisted out of the loop. If condition 2 fails and the operation is a load, it still might be possible to hoist the operation if a register can be allocated to the loaded value for the duration of the loop.

4.1.2 Load-After-Load Optimization

The load-after-load optimization is applied in situations where a load operation accesses a memory value that has been previously loaded and no intervening modification has occurred to that location's value (i.e. there is no intermittent store). In Fig. 5 the load-after-load optimization is found. The idea behind this optimization is to insert move operations into nodes in $mem\_op's$ latency field which will transfer the value without re-loading it. As a matter of correctness, move operations are only inserted into the nodes in $mem\_op's$ latency field if $op's$ definition reaches those nodes. Finally, *op* is deleted from the program graph and the local information is updated.

```
function remove_inv_mem_op(op, from_step)
begin
   /* Conditions necessary for hoisting: */
   /* 1. from_step must dominate all exit nodes. */
   /* 2. Only one definition exists. */
   /* 3. No other defs reach users. */
   /* 4. (stores) Defs of argument are same at loop exits. */
   if (/* conditions met */) then
      /* Move op to pre-loop steps if it's a load */
      /* and all post-loop steps if it's a store. */
      return REMOVED
   end if
   return NO_OPT
end remove_inv_mem_op

function do_load_load_opt(op, mem_op)
begin
   /* set field to the nodes at the latency of mem_op */
   foreach node in field
      if (/* node is reachable by op */) then
         /* Create move from mem_op's arg to the */
         /* arg of op. Add this move to node. */
      end if
   /* Delete op and update necessary information. */
   Return REMOVED
end do_load_load_opt
```

Figure 5: Supporting removal routines.

Although move operations are introduced into the schedule, the number of registers used does not increase (a proof appears in [9]).

4.1.3 Load-After-Store Optimization

The load-after-store optimization is used to remove a load operation which accesses a value that a store operation previously wrote to the memory. Due to limited resources it is possible that this optimization cannot be applied. Consider the partial code fragment:

Step 1:	$a[i] := b$	$b := \alpha$
Step 2:	$c := a[i]$	

To eliminate the load $c := a[i]$, and replace it with the move operation $c := b$ in step 2 would violate program semantics because it introduces a *read-wrong* conflict. The move operation must be placed in step 1 to guarantee correct results. However, in this code fragment:

Step 1:	$a[i] := b$	$c := \gamma$
Step 2:	$c := a[i]$	$d := f(c)$

placing a move operation in step 1 will violate program semantics because it introduces a *write-live* conflict—the move must be inserted into step 2. Notice that in both cases, the transformation is still possible, analysis is required to determine which step is applicable.

```
function try_load_store_opt(op, from_step, mem_op, to_step)
begin
   node = from_step
   if (/* there is a read-wrong conflict */) then
      node = to_step
   end if
   if (/* there is a write-live conflict */) then
      if (/* free cell exists*/) then
         /* Create move of mem_op's arg to free cell. */
         /* Add move op to to_step. */
         /* Create move of free cell to op's arg. */
         /* Add move op to from_step */
      else
         return NO_OPT
   else
      /* Create move of mem_op's arg to op's arg. */
      /* Add move op to node */
   end if
   /* Delete op and update necessary information. */
   return REMOVED
end try_load_store_opt
```

Figure 6: Load-After-Store Algorithm.

This optimization might not be feasible in the following situation:

Step 1:	$a[i] := b$	$b := \alpha$	$c := \gamma$
Step 2:	$c := a[i]$	$d := f(c)$	

Semantics are violated by placing $c := b$ into either time step. However, if a free storage cell exists, then the optimization can be done:

Step 1:	$a[i] := b$	$b := \alpha$	$c := \gamma$	$e := b$
Step 2:	$c := e$	$d := f(c)$		

Therefore, the precise case when the load-after-store optimization fails to remove a redundant load is composed of three conditions:

1. A move in this step results in a *read-wrong.*
2. A move in the previous step results in a *write-live.*
3. No free storage cell exists in the previous time step.

In practice, this situation occurs very infrequently.

The load-after-store optimization algorithm is found in Fig. 6. This algorithm determines which step to place a move operation. Initially, the step that *op* is in is tried. If a *read-wrong* conflict occurs, the previous step is tried. If a *write-live* conflict arises, a free cell is necessary to transfer the value. In this case, two move operations are added to the schedule. If a free cell is not available, no optimization is done. If no conflicts occur (or they can be alleviated by switching steps) then a move operation is inserted. Finally, the load operation is deleted and necessary information updated.

4.2 Example

Applying our transformation to the behavior in the introduction will eliminate the loads of *b[i][j-1]* and *b[i][j-2]* and the invariant load and store to *a[i]*. During loop pipelining, the load of *b[i][j-2]* for iteration *j+1* is the same as *b[i][j-1]* from iteration *j* since $(j+1)-2 = j-1$. For the invariants the store cannot be removed unless the load is also removed. Once the load is hoisted, the store can then be hoisted as well.

5 Experiments and Results

Four memory-intensive benchmarks were used to study our transformation: three numerical algorithms (prefix sums, tri-diagonal elimination and general linear recurrence equations) which are core routines in many algorithms (as discussed in the introduction) adapated from [10] and a two-dimensional hydrodynamics implicit computation adapted from [20].

Latencies used for scheduling these behaviors were two steps for add/subtract, three steps for multiply, and five steps for load/store. Also, the memory model adopted here assumed that:

- memory ports are homogenous,
- each port has its own address calculator,
- the memory is pipelined with no bank conflicts.

With these assumptions, two experiments were conducted. In the first, schedules were generated with the number of memory ports constrained between one and four and *no* functional unit (FU) constraints. Two schedules were produced for each benchmark with the sole difference between them the application of our transformation. The goal of this experiment was to isolate the difference in transformed schedules without the bias of FU constraints. In the second experiment, schedules were generated with one to four memory ports, two adder units and one multiplier unit. This experiment was designed to study performance in the presence of realistic FU resources.

For each experiment, the number of steps in the schedule of the innermost loop was counted. The GLR equations benchmark (marked with a $\star$) has two loops at the same innermost nesting level; the results indicate the summation of the number of steps in both loops. The results of experiments one and two are found in Tables 1 and 2, respectively. The column labelled "RE" indicates application of our transformation. The columns collectively labelled "Number of Ports" contain the number of steps in the innermost loop for the respective FU and memory port parameters.

The results for experiment one (Table 1) demonstrate that this optimization considerably reduces the number of cycles for the inner loop. In the prefix sums and tri-diagonal elimination benchmarks, a performance limited by the latency of a load is achieved with a sufficient number of ports. Since a latency of 5 cycles was used for load operations and not all loads can be eliminated, the schedule length cannot be any shorter. The same characteristic is exhibited by the GLR equations benchmark, although computational latency causes a longer schedule length while the hydrodynamics benchmark exhibits im-

Benchmark	RE	Number of Ports			
		1	2	3	4
Prefix Sums (scan)	no	8	8	8	8
	yes	6	5	5	5
Tri-diag. Elim. (Below Diag.)	no	10	9	9	9
	yes	7	6	5	5
GLR equations⋆	no	25	24	24	24
	yes	14	12	12	12
2D-Hydrodynamics (implicit)	no	32	28	26	25
	yes	26	24	24	22

Table 1: Steps for schedules with unlimited FUs.

Benchmark	RE	Number of Ports			
		1	2	3	4
Prefix Sums (scan)	no	9	9	9	9
	yes	6	5	5	5
Tri-diag. Elim. (Below Diag.)	no	10	9	9	9
	yes	7	6	5	5
GLR equations⋆	no	28	28	28	28
	yes	17	16	16	16
2D-Hydrodynamics (implicit)	no	38	32	28	26
	yes	26	25	24	23

Table 2: Steps for schedules with resource constraints.

proved performance as the number of memory ports increases.

The results of experiment two (Table 2) indicate that load elimination plays an important role in the presence of realistic resource constraints. For the prefix sums and tri-diagonal elimination benchmarks, the transformed behaviors were not affected by the resource constraints due to the increased flexibility in scheduling. When a load operation is removed, the dependent operations can move to earlier time steps. In resource constrained scheduling these operations have a much higher mobility, and thus a higher degree of scheduling freedom, w.r.t. the untransformed behavior. This flexibility is also demonstrated by the GLR equations and hydrodynamics benchmarks, although they are also affected by the particular resource constraints.

6 Conclusion

In this paper we have presented a new local scheduler transformation which optimizes the accessing of a secondary memory thereby reducing memory traffic. This method is based on the redundancy found in memory access instructions both within and across iterations of a loop. The method of eliminating these redundancies is through the use of memory aliasing theory, which determines when two memory instructions access the same location. With this foundation, our technique provides a powerful method of memory optimization in contrast to a simplistic pattern-matching approach (either the comparison of source-level text or the overlaying of the behavioral CDFG to determine equivalency) that can often and easily be fooled. We have presented our algorithm in detail and provided results of its application to several benchmarks which demonstrate the utility and power of this memory minimization transformation. In this paper we have restricted our discussion to memory traffic minimization. We believe that this transformation, when used in conjunction with other traditional HLS transformations, should yield better designs for memory-intensive applications. Future work will address this interaction.

References

[1] A. H. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Klewer Academic Publishers, Norwell, MA, 1993.

[3] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. *Proceedings of ICPP*, 1987.

[4] R. Camposano. Path-Based Scheduling for Synthesis. *IEEE Trans. on CAD*, 10(1), 1991.

[5] C. Chu, M. Potkonjak, M. Thaler, and J. Rabey. HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications. *ICCD-89*, 1989.

[6] F. Franssen, M. van Swaaij, F. Catthoor, and H. De Man. Modeling Piece-wise Linear and Data dependent Signal Indexing for Multi-dimensional Signal Processing. *6th International Workshop on High-Level Synthesis*, November 1992.

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

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

[9] D. J. Kolson, A. Nicolau, and N. Dutt. Minimization of Memory Traffic in High-Level Synthesis. Technical Report 93-46, U.C. Irvine, October 1993.

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

[11] P. E. R. Lippens, J. L. van Meerbergen, W. F. J. Verhaegh, and A. van der Werf. Allocation of Multiport Memories for Hierarchical Data Streams. *ICCAD-93*, 1993.

[12] A. Nicolau. *Parallelism, Memory Anti-Aliasing and Correctness for Trace-Scheduling Compilers.* PhD thesis, Yale University, March 1985.

[13] K. O'Brien, M. Rahmouni, and A. Jerraya. DLS: A Scheduling Algorithm for High-Level Synthesis in VHDL. *EDAC-93*, 1993.

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

[15] N. Park and A. C. Parker. Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications. *IEEE Trans. on CAD*, 7(3), 1988.

[16] P. Pöchmüller, M. Glesner, and F. Longsen. High-Level Synthesis Transformations for Programmable Architectures. *Euro-DAC '93*, 1993.

[17] R. Potasman, J. Lis, A. Nicolau, and D. Gajski. Percolation Based Synthesis. *27th ACM/IEEE DAC*, 1990.

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

[19] C. B. Shung et al. An Integrated CAD System for Algorithm-Specific IC Design. *IEEE Trans. on CAD*, April 1991.

[20] 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.